A NANOPASS FRAMEWORK FOR COMMERCIAL COMPILER DEVELOPMENT

Andrew W. Keep

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing
Indiana University
December 2012

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

R. Kent Dybvig, Ph.D.

Daniel P. Friedman, Ph.D.

Ryan Newton, Ph.D.

Markus Dickinson, Ph.D.

12/17/2012

Copyright 2012
Andrew W. Keep
All rights reserved

Acknowledgements

First, I would like to thank my family. They have supported me throughout my graduate school process from my initial application, through the challenges of graduate school, and the final dissertation process. I could not have done this without their love and support. My mother, Susan Keep, has encouraged me when I needed it most and helped remind me that I had the skills and determination I needed to make it through the process. I cannot thank her enough for all of her support (and for her help editing this dissertation). My father, Bill Keep, helped me focus on the end goal, even when it seemed furthest away, and reminded me that this is a training process and I have many years ahead of me to expand my research. My brother, Ben Keep, has been a great friend and confidant throughout this process, and I really appreciate the conversations we had along the way, both those related to graduate school and those that helped remind me of the broader world.

This dissertation would not have been possible without my advisor R. Kent Dybvig. I have had the pleasure of working closely with him as his assistant instructor, his employee, his fellow programmer, and his friend. I have improved my programming skills, learned how to do and talk about my research, and had many conversations about career, academia, and life in general. I would not trade the time we spent working together for the world.

I would like to thank my dissertation committee, Dan Friedman, Ryan Newton, and Markus Dickinson. My first programming languages class at Indiana University was Dan's Principles of Programming Languages course, and it helped set the path for my research at Indiana University. Ryan and I have had some interesting conversations in the short time we have had to work together. Markus has been kind enough to serve on my committee as the representative from my Linguistics Minor.

I would like to thank Dipanwita Sarkar for developing the prototype nanopass framework and allowing me to use it as a starting point for my own research on nanopass frameworks.

I would like to thank Geoffrey Brown and David Wise for giving me my first opportunity to teach a core Computer Science course my first semester at Indiana University.

I would like to thank Arun Chauhan for getting me started in compiler research and compiler framework building.

I would like to thank my fellow students and friends in the Computer Science program at Indiana University, especially those in the Programming Languages group, PL-wonks, for many interesting conversations, their support, and their friendship over the last five and half years.

Finally, it is hard to imagine how this project would have been completed without the support I received from Cisco Systems, Inc., first as an employee of Cadence Research Systems working on a contract for Cisco and later as a part-time intern. I had the opportunity to work with a team of talented people on an interesting project that I am sad to leave. I wish them future success on the project and all the best.

A NANOPASS FRAMEWORK FOR COMMERCIAL COMPILER DEVELOPMENT

Andrew W. Keep

Contemporary commercial compilers typically handle sophisticated high-level source languages, generate efficient assembly or machine code for multiple hardware architectures, run under and generate code to run under multiple operating systems, and support source-level debugging, profiling, and other program development tools. As a result, commercial compilers tend to be among the most complex of software systems.

Nanopass frameworks are designed to help make this complexity manageable. A nanopass framework is a domain-specific language, embedded in a general purpose programming language, to aid in compiler development. A nanopass compiler is comprised of many small passes, each of which performs a single task and specifies only the interesting transformations to be performed by the pass. Intermediate languages are formally specified by the compiler writer, which allows the infrastructure both to verify that the output of each pass is well-formed and to fill in the uninteresting boilerplate parts of each pass.

Prior nanopass frameworks were prototype systems aimed at educational use, but we believe that a suitable nanopass framework can be used to support the development of commercial compilers. We have created such a framework and have demonstrated its effectiveness by using the framework to create a new commercial compiler that is a "plug replacement" for an existing commercial compiler. The new compiler uses a more sophisticated, although slower, register allocator and implements nearly all of the optimizations of the original compiler, along with several "new and improved" optimizations. When compared to the original compiler on a set of benchmarks, code generated by the new compiler runs, on average, 21.5% faster. The average compile time for these benchmarks is less than twice as long as with the original compiler. This dissertation provides a description of the new framework, the new compiler, and several experiments that demonstrate the performance and effectiveness of both, as well as a presentation of several optimizations performed by the new compiler and facilitated by the infrastructure.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Compilers are traditionally structured as series of *passes*, each of which reads an input file and produces an output file. This was originally necessary because an entire source program, or a representation of it, could not generally fit in memory. To avoid excessive overhead for reading and writing intermediate files, it was desirable to minimize the number of passes [**8**]. Contemporary commercial compilers, however, are expected to generate efficient code for multiple architectures, run under multiple operating systems, and support source-level debugging, profiling, and other program development tools. A compiler that meets these expectations with a minimal number of passes is likely to be complex and difficult to modify or extend. For example, it might be difficult or impossible to add a new transformation that must run between two existing transformations performed by the same pass.

Fortunately, contemporary hardware systems have vastly larger memories and even larger virtual address spaces so that, while compilers are still structured as series of passes, intermediate representations can be maintained in memory, which obviates the intermediate files and greatly reduces the incentive to minimize the number of passes. Thus, a natural way to manage the complexity of a contemporary compiler is to break up the compiler into many smaller passes, just as one would modularize any complex program as a means to make it more manageable.

A nanopass infrastructure is intended to support taking the above approach to its natural extreme, in which a compiler is structured as a series of many small passes. A nanopass framework is an embedded domain specific language (DSL) that supports compiler development. A nanopass compiler is comprised of many small passes, each of which performs a single task and specifies only the interesting transformations to be performed by the pass. The compiler writer formally specifies each intermediate language over which a pass operates. The nanopass framework uses this information to verify that the output of each pass is well-formed and to fill in the uninteresting, boilerplate parts of each pass.

In 2004, Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig submitted a paper to the International Conference on Functional Programming (ICFP) [**74**] that positioned a prior nanopass

framework as a tool for both educational use and for the development of commercial compilers. The reviewers accepted the paper, but only after it was repositioned to describe the nanopass framework as a tool for educational use only. They did not believe that the nanopass framework could be used to write a commercial compiler, and, in retrospect, they were correct. The nanopass framework developed by Sarkar et al. demonstrated the viability of a nanopass framework but was never fully tested, even in an educational setting.

We believe that a suitably improved nanopass framework can be used to support the development of commercial compilers. To this end, we devised a research plan to improve the prototype nanopass framework and to build a new commercial compiler with it. We have improved the prototype framework and have demonstrated its effectiveness by using the framework to create a new commercial compiler that is a "plug replacement" for the commercial Chez Scheme [**33**] compiler for the Scheme programming language [**34**, **83**]. The Chez Scheme compiler was designed to generate efficient code and is almost *absurdly* fast. The goal for the new compiler is to generate code that is on par with the code generated by the existing Chez Scheme compiler and to do so with compile time that is within a factor of two of the existing compiler. The extra compile time allows us to experiment with a more sophisticated, although slower, register allocator. The new compiler also implements nearly all of the optimizations of the original compiler, along with several "new and improved" optimizations.

The new compiler meets the goals set out in the research plan. When compared to the original compiler on a set of benchmarks, the benchmarks, for the new compiler run, on average, between 15.0% and 26.6% faster, depending on the architecture and optimization level. The compile times for the new compiler are also well within the goal, with a range of 1.64 to 1.75 times slower.

We have also developed a nanopass version of the student compiler used in a course on compiler design and implementation. The new student compiler was implemented over the course of a weekend, based on the existing student-compiler source code. The nanopass version of the class compiler runs a small test suite 52% faster, and shrinks the source code for passes by 21% by eliminating boilerplate code.

The specification for the new nanopass framework is described in Chapter 2, using examples from the class compiler. Chapter 2 also presents the history of nanopass frameworks and a comparison of the new nanopass framework with the one developed by Sarkar et al., upon which the new framework is based.

The new nanopass framework is evaluated in Chapter 3. This chapter starts with a comparison of the two versions of the student compiler and then presents a comparison of the original and nanopass versions of the Chez Scheme compiler. The two versions of the Chez Scheme compiler have a similar

front end but differ significantly in the back end. These differences are described, and two of the front-end passes are compared to illustrate the overhead in the use of the nanopass framework, when compared with the original internal representation. This chapter also presents a comparison of the performance of a set of benchmarks compiled with the two versions of the Chez Scheme compiler. The compile time of the two compilers is also compared. The compile time is broken down into front-end and back-end times to determine the source of compile-time overhead.

The new compiler performs implicit cross-library optimization in addition to supporting the original compiler's `library-group` form. Normally, each Revised^6 Report on Scheme (R6RS) library or top-level program [**83**] would be a separate compilation unit, with no optimizations across the library boundaries. The `library-group` form allows several R6RS libraries and optionally an R6RS top-level program to be combined into a single compilation unit. This allows optimization to occur across the library boundaries, as though the libraries were included in a single source file. The implicit cross-library optimizations are not as general as the `library-group` form but allow for some constant propagation and procedure inlining across library boundaries, even when the `library-group` form is not used. Chapter 4 of this dissertation contains a description of the `library-group` form and the implicit cross-library optimizations.

The new compiler also supports an improved and simplified version of the closure optimizations found in the original Chez Scheme compiler. The new closure optimization statically justifies the removal of 56.94% of closures and eliminates 44.89% of the total free-variables, which results in 58.25% less memory allocation and 58.58% fewer memory references due to closures. This optimization is among the reasons that the new compiler performs better on the benchmarks. Chapter 5 of this dissertation provides a description of the closure conversion algorithm and its implementation in detail.

The new nanopass framework and the new Chez Scheme compiler demonstrate that the nanopass framework is a viable tool for writing commercial compilers. The experience of working on the new compiler also showed areas into which the nanopass framework could be extended to make commercial compiler development even easier. Chapter 6 presents a discussion of this future work and concludes the dissertation.

Background

While a nanopass framework could be developed in almost any programming language, both the prototype nanopass framework and the new nanopass framework are embedded in Scheme and implemented using the `syntax-case` macro expander [**31**, **40**]. Some familiarity with the Scheme programming language is necessary to understand the description and examples in Chapter 2. A

passing knowledge of Scheme macros and macro expansion is also useful to understand the description of the syntactic forms provided by the nanopass framework.

The description of the nanopass framework in Chapter 2 also assumes some familiarity with the `match` form. The `match` form is a Scheme extension used to pattern match S-expressions. For example, we could use match to extract the numbers from `(a 1 2 3 4)` as follows:

```
(match '(a 1 2 3 4)
  [(a ,x0 ,x1 ...) (list x1 x0)])  ⇒  ((2 3 4) 1)
```

The pattern `(a ,x0 ,x1 ...)` matches a list that starts with the symbol `a` and contains one or more additional items, where the first item is bound to `x0` and the remaining items are bound in a list to `x1`. The `unquote` (`,`) in the pattern indicates that `x0` and `x1` are pattern variables and the ellipsis (...) indicates that zero or more items should be matched by the preceding pattern, in this case `,x1`.

In the context of the micropass compiler described in Chapter 2, which inspired the nanopass framework, `match` can be used to perform term rewriting, where an input-language S-expression is matched and an output-language S-expression is constructed. For example, we could write a pass to remove one-armed `if` and replace the multi-expression body of `lambda` and `letrec` from the following language:

```
e,body ∈ Expr  ⟶  x
               |  (quote d)
               |  (if e0 e1)
               |  (if e0 e1 e2)
               |  (begin e* ... e)
               |  (lambda (x* ...) body* ... body)
               |  (letrec ([x* e*] ...) body* ... body)
               |  (e e* ...)
```

where x is a symbol and represents a variable reference, and d represents a Scheme datum, to the language:

```
e,body ∈ Expr  ⟶  x
               |  (quote d)
               |  (if e0 e1 e2)
               |  (begin e* ... e)
               |  (lambda (x* ...) body)
               |  (letrec ([x* e*] ...) body)
               |  (e e* ...)
```

The pass could be written using match as follows:

```
(define make-begin
  (lambda (e* e)
    (if (null? e*)
        e
        `(begin ,@e* ,e))))

(define simplify
  (lambda (x)
    (match x
      [,x (guard (symbol? x)) x]
      [(quote ,d) `(quote ,d)]
      [(if ,e0 ,e1)
       (let ([e0 (simplify e0)] [e1 (simplify e1)])
         `(if ,e0 ,e1 (void)))]
      [(if ,e0 ,e1 ,e2)
       (let ([e0 (simplify e0)] [e1 (simplify e1)] [e2 (simplify e2)])
         `(if ,e0 ,e1 ,e2))]
      [(begin ,e* ... ,e)
       (let ([e* (map simplify e*)] [e (simplify e)])
         (make-begin e* e))]
      [(lambda (,x* ...) ,body* ... ,body)
       (let ([body* (map simplify body*)] [body (simplify body)])
         `(lambda (,x* ...) ,(make-begin body* body)))]
      [(letrec ([,x* ,e*] ...) ,body* ... ,body)
       (let ([e* (map simplify e*)]
             [body* (map simplify body*)]
             [body (simplify body)])
         `(letrec ([,x* ,e*] ...) ,(make-begin body* body)))]
      [(,e ,e* ...)
       (let ([e (simplify e)] [e* (map simplify e*)])
         `(,e ,e* ...))])))
```

Here, the `make-begin` helper builds a `begin` form when there is more than one expression in the body of a `begin`, `lambda`, or `letrec`. The `simplify` procedure matches each of the forms of the input language, recurs through the sub-expressions of each form, and constructs a new output term in the output language.

The `match` form also supports catamorphisms [**68**] to recur through the sub-expressions of the input forms. A catamorphism, for our purposes, recurs through sub-forms in the language until a terminal case, such as `x` or `(quote` d`)`, is found. The `simplify` pattern can be rewritten to use catamorphisms as follows:

```
(define simplify
  (lambda (x)
```

```
(match x
  [,x (guard (symbol? x)) x]
  [(quote ,d) `(quote ,d)]
  [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]
  [(if ,[e0] ,[e1] ,[e2]) `(if ,e0 ,e1 ,e2)]
  [(begin ,[e*] ... ,[e]) (make-begin e* e)]
  [(lambda (,x* ...) ,[body*] ... ,[body])
   `(lambda (,x* ...) ,(make-begin body* body))]
  [(letrec ([,x* ,[e*]] ...) ,[body*] ... ,[body])
   `(letrec ([,x* ,e*] ...) ,(make-begin body* body))]
  [(,[e] ,[e*] ...) `(,e ,e* ...)]))))
```

Here, the square brackets (`[ ]`) in the syntax `,[e0]` indicate that a catamorphism should be applied. The version of the `simplify` procedure using catamorphisms implements the same algorithm as the preceding `simplify` procedure but is more succinct. The `match` form's syntax is the inspiration for the S-expression pattern matching and templates in the nanopass framework.

This dissertation also describes several techniques common to compiler implementation and optimization. The examples and descriptions of the nanopass framework in Chapter 2 and the description of the original and new Chez Scheme compilers in Chapter 3 mention a few algorithms the reader should be familiar with.

The first is *free-variable analysis*, in which the free variables of a procedure are identified. A *free variable* is a variable not bound as a formal or through a local binding form, such as `let` or `letrec`, within the body of a procedure. Free-variable analysis traverses the body of each procedure to gather the set of referenced and assigned variables. This process proceeds inner-most to outer-most, with variables removed from the free-variable set as the binding form that names them is found. When a λ-expression is found, the free-variable set is recorded for this λ-expression.

In languages with higher-order procedures, such as Scheme, ML, Haskell, or JavaScript, the free-variable set for each λ-expression can be used for *closure conversion*. A *closure* is a first-class object that encapsulates some representation of a procedure's code (e.g., the starting address of its machine code), along with some representation of the lexical environment. A closure-conversion algorithm makes the representation of a closure explicit. Our closure-conversion algorithm, discussed in detail in Chapter 5, uses the results of free-variable analysis to determine what values should be stored in the closure for each procedure.

In a compiler, some passes are said to be *flow sensitive* when they must follow the control flow of a program or procedure. A *forward-flow* analysis is a flow-sensitive analysis that begins at the entry point to a program or procedure and follows the control flow to the exit point of the program or

procedure. A *reverse-flow* analysis is a flow-sensitive analysis that begins at the exit point of a program or procedure and proceeds in reverse order up the control flow to the entry point of the program or procedure. A *flow-insensitive* pass is not constrained by the control flow of a program or procedure.

Chapter 3 also discusses *register allocation*. Register allocation is the process by which program variables and compiler-introduced temporaries are allocated to register or frame-location homes. This process is complicated because CPUs have a limited number of registers, e.g., just 8 general purpose registers on an x86 chip and just 16 on an x86_64 chip, but all of the variables in a program must either be located in a register or in memory. Because registers provide faster access and mutation, it is generally desirable to place variables in registers. In some cases, operand constraints imposed by the target machine may require that operands be placed in registers. The process of meeting machine constraints is called *instruction selection*, and we consider this to be part of the register allocation process. Instruction selection chooses the machine instructions to perform an operation and enforces the operand constraints for each instruction, introducing *unspillable* variables if needed. An unspillable variable is a variable with a short lifetime that cannot be spilled to the frame.

The original Chez Scheme compiler uses a simple linear register allocator with a lazy save and restore strategy [**21**] for saving variables around non-tail calls. This register allocator tracks the *liveness* of registers by processing the body of each procedure in a reverse-flow analysis. A variable, register, or frame location is live when the value it contains might still be referenced. When the value of a variable, register, or frame location is set it is said to be *killed*. A *live analysis* performs a reverse-flow analysis to determine when variables, registers, or frame-locations are live.

The new Chez Scheme compiler uses a graph-coloring register allocator [**17**]. A graph-coloring register allocator first constructs a *conflict graph* using live analysis and then attempts to "color" each node of the graph in a way that no two nodes that share an edge have the same color. The conflict graph records the conflicts between variables and registers (or between variables and frame variables when the related *frame allocation* is performed). A variable `x` conflicts with a variable, register, or frame-location `y` when `x` and `y` are live at the same time and might hold different values. When a graph cannot be colored, one or more variables must be *spilled* in order to create a graph that can be colored. Spilled variables are allocated to frame locations through frame allocation. After frame allocation, instruction selection is performed to ensure that spilled variables do not cause the instructions to violate machine operand constraints.

Finally, Chapter 3 briefly describes the source optimizer [**92**]. The source optimizer performs bounded, aggressive procedure *inlining* along with constant propagation, copy propagation, and constant folding. Inlining copies a procedure to the position where it is called. Constant propagation is an optimization that replaces the references to a variable with a constant, when the variable is bound to a constant and is unassigned between the binding and the reference. Copy propagation is an optimization that replaces references to a variable with a second variable when the first variable is bound to the second, and the variables are unassigned. Constant folding is an optimization that allows a computation that would normally occur at run time to be done at compile time when the operation is known and the operands are constant. Together these three optimizations help to reduce the increase in size due to inlining.

Additional information about compiler development in Scheme can be found in the description of an earlier compiler course [**61**] by Hilsdale et al. and in the description of building a compiler from scratch [**50**] by Ghuloum. Also, the reader may find the following textbooks useful: for a general knowledge of compiling functional programming languages, *Compiling with Continuations* [**10**]; and for a general knowledge of compiler techniques and terms, *Compilers: Principles, Techniques, and Tools* [**7**] by Aho et al.

CHAPTER 2

The Nanopass Framework

2.1. Introduction

The idea of writing a compiler as a series of small, single-purpose passes grew out of a course on compiler construction taught by Dan Friedman in 1999 at Indiana University. The following year, R. Kent Dybvig and Oscar Waddell joined Friedman to refine the idea of the *micropass compiler* into a set of assignments that could be used in a single semester to construct a compiler for a subset of Scheme. The micropass compiler uses an S-expression pattern matcher developed by Friedman to simplify the matching and rebuilding of language terms. Erik Hilsdale added a support for catamorphisms [**68**] that provides a more succinct syntax for recurring into sub-terms of the language, which further simplified pass development.

Passes in a micropass compiler are easy to understand, as each pass is responsible for just one transformation. The compiler is easier to debug when compared with a traditional compiler composed of a few, multi-task passes. The output from each pass can be inspected to ensure that it meets grammatical and extra-grammatical constraints. The output from each pass can also be tested in the host Scheme system to ensure that the output of each pass evaluates to the value of the initial expression. This makes it easier to isolate broken passes and identify bugs. The compiler is more flexible than a compiler composed of a few, multi-task passes. New passes can easily be added between existing passes, which allows experimentation with new optimizations. In an academic setting, writing compilers composed of many, single-task passes is useful for assigning extra compiler passes to advanced students who take the course.

Micropass compilers are not without drawbacks. First, efficiency can be a problem due to pattern-matching overhead and the need to rebuild large S-expressions. Second, passes often contain boilerplate code to recur through otherwise unchanging language forms. For instance, in a pass to remove one-armed `if` expressions, where only the `if` form changes, other forms in the language must be handled explicitly to locate embedded `if` expressions. Third, the representation lacks formal structure.

The grammar of each intermediate language can be documented in comments, but the structure is not enforced.

A nanopass framework addresses these problems with two syntactic forms: `define-language` and `define-pass`. A `define-language` form formally specifies the grammar of an intermediate language. A `define-pass` form defines a pass that operates on one language and produces output in a possibly different language. Formally specifying the grammar of an intermediate language and writing passes based on these intermediate languages allows the nanopass framework to use a record-based representation of language terms that is more efficient than the S-expression representation, autogenerate boilerplate code to recur through otherwise unchanging language forms, and generate checks to verify that the output of each pass adheres to the output-language grammar.

The summer after Dybvig, Waddell, and Friedman taught their course, Jordan Johnson implemented an initial prototype of the nanopass framework to support the construction of micropass compilers. In 2004, Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig developed a more complete prototype nanopass framework for compiler construction and submitted a paper on it to ICFP [**74**]. The initial paper focused on the nanopass framework as a tool capable of developing both academic and commercial quality compilers. The paper was accepted but on the condition that it be refocused only on academic uses. The reviewers were not convinced that the framework or nanopass construction method was capable of supporting a commercial compiler. In retrospect, the reviewers were right. Sarkar implemented only a few of the passes from the compiler used in the course on compilers. This implementation showed that the nanopass framework was viable, but it did not support the claim that the nanopass framework could be used for a commercial compiler. In fact, because the class compiler was started but never completed, it is unclear whether the prototype was even up to the task of writing the full class compiler.

The nanopass framework described in this dissertation improves on the prototype developed by Sarkar. In this framework, language definitions are no longer restricted to top-level definitions. Additionally, passes can accept more than one argument and return zero or more values. Passes can be defined that operate on a subset of a language instead of being restricted to starting from the entry-point nonterminal of the language. Passes can also autogenerate nonterminal transformers not supplied by the compiler writer. The new nanopass framework also defines two new syntactic forms, `nanopass-case` and `with-output-language`, that allow language terms to be matched and constructed outside the context of a pass.

Although the nanopass framework defines just two primary syntactic forms, the macros that implement them are complex, with approximately 4600 lines of code. In both the prototype and the new version of the nanopass framework, the `define-language` macro parses a language definition and stores a representation of it in the compile-time environment. This representation can be used to guide the definition of derived languages and the construction of passes. Both also create a set of record types used to represent language terms at run time, along with an unparser for translating the record representation to an S-expression representation. Finally, both create meta-parsers to parse S-expression patterns and templates.

The `define-pass` form, in both versions of the framework, operates over an input-language term and produces an output-language term. The input-language meta-parser generates code to match the specified pattern as records, as well as a set of bindings for the variables named in the pattern. The output-language meta-parser generates record constructors and grammar-checking code. Within a pass definition, a transformer is used to define a translation from an input nonterminal to an output nonterminal. Each transformer has a set of clauses that match an input-language term and construct an output-language term. The pattern matching also supports catamorphisms [**68**] for recurring into language sub-terms.

2.2. The Prototype Framework

The prototype nanopass framework developed by Sarkar [**73**] was an important first step toward the nanopass framework described in this dissertation. The prototype was designed with the class compiler in mind and provided a basis for the current nanopass framework.

The prototype provides two syntactic forms, `define-language` and `define-pass`. The framework also provides a `language->s-expression` form that returns a fully specified language definition for a named language. This can be helpful when working with a language that is defined as the extension of a base language, particularly when the base language is also an extended language.

The `define-language` form allows the compiler writer to specify a set of terminals and nonterminals for the language. It also includes a `definitions` clause to specify a set of definitions to be included when a language term is unparsed. The `definitions` clause is intended to allow an unparsed language to be run in the host Scheme system. The `define-language` syntax has two main forms, the full language definition form that fully specifies a language and an extended language form that allows the compiler writer to specify what should be removed from and added to a base language to define a language with the desired grammar.

The `define-language` form has three programmer-visible outputs: a *parser* to parse an S-expression representation into a record representation; an *unparser* to create an S-expression representation from the record representation; and the *language definition*, a compile-time value associated with the identifier used to define the language. The compile-time language specification is represented both (1) as an S-expression that lists the terminals and nonterminals; and (2) as an annotated version of this representation that also includes the record names, record predicates, and record accessors for the records used to represent language terms.

In addition to these visible products, `define-language` creates a set of record definitions to represent intermediate-language programs in the language. The `define-language` form also creates a meta-parser that the `define-pass` macro uses to parse S-expression patterns into a set of record predicate checks as well as templates into a set of record constructor calls and argument-type checks. The meta-parser is defined as a meta-definition, i.e., an ordinary Scheme function that can be called only during expansion. This limits where `define-language` can be used, as the meta-definition must be visible to the Scheme `eval` procedure, which effectively requires that the language definition be at the top level of a program or library.

The `define-pass` form specifies a function for operating over an input language and constructs a term in a (possibly different) output language. The `define-pass` form has three main parts: a `definitions` clause that contains Scheme definitions in the same scope as the transformer clauses, a set of transformer clauses for operating over the nonterminals of the input language, and a restricted body form. The `definitions` clause is optional. The body is restricted to either an empty list or a `let-values` expression that binds the return values of a transformer and returns the first return value, which is the language term. When the empty list is specified as the body, the first transformer in the pass is considered the entry transformer, the transformer is automatically called, and its first return value is returned by the pass.

Each transformer in `define-pass` has a signature that includes a name, the input-language nonterminal, a list of additional arguments with initial values, an output-language nonterminal, and a list of procedures for combining extra return values for use in autogenerated clauses. A set of user-specified clauses follows the transformer signature. Each clause uses an S-expression pattern to match the incoming language term, an optional guard clause, and a right-hand-side expression that produces the output-language expression.

The pattern supports a catamorphism syntax for recurring on sub-nonterminals of the matched production. The catamorphism syntax requires a binding for the output-language term and each

extra return value produced by the recursive call. Optionally, the input-language term can also be bound so that it can be referred to in the guard. The target of the catamorphism can be optionally specified as a procedure expression. When the target is not specified, the input type, the output type, and the number of extra return values expected are compared to find matching transformers. If exactly one transformer is found, it is used as the target for the catamorphism; otherwise, a syntax violation occurs. When the catamorphism is used to call a transformer that expects additional arguments, the initial values specified in the target transformer signature are used.

Within the right-hand-side expression, `quasiquote` is bound to a macro that constructs terms in the output language for the nonterminal specified by the transformer signature. To construct terms for other nonterminals, an additional form, `in-context`, is also bound to allow the compiler writer to specify a different nonterminal in the output language, which effectively rebinds `quasiquote`.

Within the transformer, a clause for a production can be omitted when the input-language nonterminal and output-language nonterminal both contain the same production. In fact, transformers for nonterminals that have the same set of productions in the input language and output language can omit all of the clauses, and the `define-pass` form will fill in the necessary recursive calls. Missing clauses are autogenerated to recur through any sub-nonterminal in the production before producing the appropriate output-language term. The target of the recursion is determined by looking for a transformer that expects the field type as input. This is a different procedure from the one used to find the target of a catamorphism call. When the clause supplied by the compiler writer for a production does not match the most general case of the production, the nanopass framework will autogenerate a clause for this production. This can happen, for instance, if the user-supplied pattern includes a match for a specific sub-nonterminal or if the compiler writer specifies a guard. Such clauses run only after clauses specified by the compiler writer are exhausted.

The `define-pass` form creates an internal function definition for each transformer and converts the S-expression pattern into a set of record predicate calls to determine, along with the optional guard, whether the pattern matches. Once the pattern is matched and the guard, if any, is satisfied, the catamorphism calls are performed; and the body is executed.

The prototype nanopass framework demonstrated that a nanopass framework is a viable idea. However, the prototype was never fully tested, as only the first 20 passes of the class compiler (out of about 50) were specified. The register allocator and code generator were never completed. Due to the way that the class compiler is written, the register allocator requires a predicate to determine when register allocation is complete, and the code generator requires a pass that has no output value

and emits the assembly language for the program to the standard output port. While some work had been done to allow passes with no output language, this feature was never tested and using it leads to errors. There was no direct way to write the predicate for the register allocator. In both cases, these limitations could be worked around but not without the pass doing extra work to produce a language term that would be discarded.

2.3. New Framework Specification

The new nanopass framework builds on the prototype described in the preceding section. The examples in this section are pulled from a nanopass version of the class compiler.

2.3.1. Defining languages. The nanopass framework operates over a set of compiler-writer-defined languages. Languages defined in this way are similar to context-free grammars, in that they are composed of a set of terminals, a set of nonterminal symbols, a set of productions for each nonterminal, and a start symbol from the set of nonterminal symbols. We refer to the start symbol as the entry nonterminal of the language. An intermediate language definition for a simple variant of the Scheme programming language, post macro expansion, might look like:

```
(define-language Lsrc
  (entry Expr)
  (terminals
    (uvar (x))
    (primitive (pr))
    (datum (d)))
  (Expr (e body)
    x
    (quote d)
    (if e0 e1 e2)
    (begin e* ... e)
    (lambda (x* ...) body)
    (let ([x* e*] ...) body)
    (letrec ([x* e*] ...) body)
    (set! x e)
    (pr e* ...)
    (call e e* ...)             => (e e* ...)))
```

The `Lsrc` language defines a subset of Scheme suitable for the compiler course. It is the output language of a more general "parser" or simple expander that parses S-expressions into `Lsrc` language forms. The `Lsrc` language consists of a set of terminals (listed in the `terminals` form) and a single nonterminal `Expr`. The terminals of the language are `uvar` (for unique variables), `primitive`, and `datum` (for the subset of Scheme datum supported by this language). The compiler writer must

supply a predicate corresponding to each terminal, lexically visible where the language is defined. The nanopass framework derives the predicate name from the terminal name by adding a `?` to the terminal name. In this case, the nanopass framework expects `uvar?`, `primitive?`, and `datum?` to be lexically visible where `Lsrc` is defined.

Each terminal clause lists one or more meta-variables, used to refer to the terminal in nonterminal productions. Here, `x` refers to a `uvar`, `pr` refers to a `primitive`, and `d` refers to a `datum`.

For our class compiler, a `uvar` is represented as a symbol with the format *name.num*. The predicate below ensures that the symbol is well-formed, with a numeric string that cannot be misinterpreted as another number, since part of the goal of the `uvar` representation is uniqueness.

```
(define uvar?
  (lambda (x)
    (and (symbol? x)
         (well-formed? x))))
```

The class compiler also selects a subset of primitives from Scheme and represents these primitives as symbols. The `primitive?` predicate is defined as follows:

```
(define primitive?
  (lambda (x)
    (memq x '(+ - * car cdr cons make-vector vector-length vector-ref void
              < <= = >= > boolean? eq? fixnum? null? pair? vector? procedure?
              set-car! set-cdr! vector-set!))))
```

The class compiler limits the Scheme datum that can be represented to constants, pairs, and vectors, where constants are limited to fixed-size integers (fixnums), `#t`, `#f`, and null. The `datum?` predicate can be defined as follows:

```
(define constant?
  (lambda (x)
    (or (boolean? x) (null? x) (fixnum? x))))

(define datum?
  (lambda (x)
    (or (constant? x)
        (and (pair? x) (datum? (car x)) (datum? (cdr x)))
        (and (vector? x)
             (let loop ([i (vector-length x)])
               (or (fxzero? i)
                   (let ([i (fx- i 1)])
                     (and (datum? (vector-ref x i))
                          (loop i)))))))))
```

The `Lsrc` language also defines the nonterminal `Expr`. Nonterminals start with a name, followed by a list of meta-variables and a set of grammar productions. In this case, the name is `Expr`, and two meta-variables, `e` and `body`, are specified. Just like the meta-variables named in the terminals clause, nonterminal meta-variables are used to represent the nonterminal in nonterminal productions. Each production follows one of three forms. It is a single meta-variable, an S-expression that starts with a keyword, or an S-expression that does not start with a keyword (referred to as an *implicit* production). The S-expression forms cannot include keywords past the initial starting keyword. In `Lsrc`, the `x` production is the only single meta-variable production and indicates that a stand-alone `uvar` is a valid `Expr`. The only implicit S-expression production is the `(pr e* ...)` production, and it indicates a primitive call that takes zero or more `Expr`s as arguments. (The `*` suffix on `e` is used by convention to indicate plurality and does not have any semantic meaning: It is the `...` that indicates that the field can take zero or more `Expr`s.) The `(call e e* ...)` production indicates a procedure call. Here, the `call` keyword is used to differentiate it from a primitive call production. The `=> (e e* ...)` syntax that follows the production indicates a *pretty* form for the production. The `define-language` form defines an unparser for each language, and the unparser uses the pretty form when unparsing this production. This is useful to produce a language form that can be evaluated in the host Scheme system. The rest of the productions are S-expression productions with keywords that correspond to the Scheme syntax that they represent.

In addition to the star, `*`, suffix mentioned earlier in the primitive call and procedure call productions, meta-variable references can also use a numeric suffix (as in the production for `if`), a question mark (`?`), or a caret (`^`). The `?` suffix is intended for use with `maybe` meta-variables, and the `^` is used when expressing meta-variables with a more mathematical syntax than the numeric suffixes provide. Suffixes can also be used in combination. References to meta-variables in a production must be unique, and the suffixes allow the same root name to be used more than once.

Language definitions can also include more than one nonterminal, as the following language illustrates:

```
(define-language L3
  (entry Expr)
  (terminals
    (constant (c))
    (primitive (pr))
    (uvar (x)))
  (Expr (e body)
    x
    (quote c)
    (if e0 e1 e2)
```

```
    (begin e* ... e)
    le
    (let ([x* e*] ...) abody)
    (letrec ([x* le*] ...) body)
    (set! x e)
    (pr e* ...)
    (call e e* ...))
  (AssignedBody (abody)
    (assigned (x* ...) body))
  (LambdaExpr (le)
    (lambda (x* ...) abody)))
```

This language has three nonterminals, `Expr`, `AssignedBody`, and `LambdaExpr`. When more than one nonterminal is specified, one must be selected as the entry point. In language `L3`, the `Expr` nonterminal is selected as the entry nonterminal by the `(entry Expr)` clause. When the entry clause is not specified, the first nonterminal listed is implicitly selected as the entry point.

The `L3` language uses a single terminal meta-variable production, `x`, to indicate that a stand-alone `uvar` is a valid `Expr`. In addition, the `L3` language uses a single nonterminal meta-variable production, `le`, to indicate that any `LambdaExpr` production is also a valid `Expr`. The `LambdaExpr` is separated from `Expr` because the `letrec` production is now limited to binding `uvar`s to `LambdaExpr`s.

In addition to the nanopass framework providing a syntax for specifying list structures in a language production, it is also possible to indicate that a field of a language production might not contain a (useful) value. The following language has an example of this:

```
(define-language Lopt
  (terminals
    (uvar (x))
    (label (l))
    (constant (c))
    (primitive (pr)))
  (Expr (e body)
    x
    (quote c)
    (begin e* ... e)
    (lambda (x* ...) body)
    (let ([x* e*] ...) body)
    (letrec ([x* le*] ...) body)
    (pr e* ...)
    (call (maybe l) (maybe e) e* ...))
  (LambdaExpr (le)
    (lambda (x* ...) body)))
```

The `(maybe l)` field indicates that either a label, `l`, or `#f` will be provided. Here, `#f` is a stand-in for bottom, indicating that the value is not specified. The `(maybe e)` field indicates that either an `Expr` or `#f` will be provided.

Instead of using `(maybe l)` to indicate a label that might be provided, a `maybe-label` terminal that serves the same purpose could be added. It is also possible to eliminate the `(maybe e)` form, although it requires the creation of a separate nonterminal that has both an `e` production and a production to represent $\perp$, when no `Expr` is available.

2.3.2. Extending languages. The first "pass" of the class compiler is a simple expander that produces `Lsrc` language forms from S-expressions. The next pass takes the `Lsrc` language and expands complex quoted datum into code appropriate to construct these constants. The output grammar of this pass changes just one production of the language, exchanging potentially complex quoted datum with quoted constants and making explicit the code to build the constant pairs and vectors when the program begins execution.

The compiler writer could specify the new language by rewriting the `Lsrc` language and replacing the appropriate terminal forms. Rewriting each language in its full form, however, can result in verbose source code, particularly in a compiler like the class compiler, which has nearly 30 different intermediate languages. Instead, the nanopass framework supports a language extension form. The output language can be specified as follows:

```
(define-language L1
  (extends Lsrc)
  (entry Expr)
  (terminals
    (- (datum (d)))
    (+ (constant (c))))
  (Expr (e body)
    (- (quote d))
    (+ (quote c))))
```

The `L1` language removes the `datum` terminal and replaces it with the `constant` terminal. It also replaces the `(quote d)` production with a `(quote c)` production to indicate that only constants are allowed in the `quote` form.[1] A language extension form is indicated by including the `extends` clause, in this case `(extends Lsrc)`, that indicates that this is an extension of the given base language. In a language extension, the `terminals` form now contains subtraction clauses, in this

[1]If we failed to replace the `(quote d)` form, it would result in an error, since the `d` meta-variable is not bound in the new language.

case `(- (datum (d)))`, and addition clauses, in this case `(+ (constant (c)))`. These addition and subtraction clauses can contain one or more terminal specifiers. The nonterminal syntax is similarly modified, with the subtraction clause, in this case `(- (quote d))`, that indicates productions to be removed and an addition clause that indicates productions to be added, in this case `(+ (quote c))`.

The list of meta-variables indicated for the nonterminal form is also updated to use the set in the extension language. It is important to include not only the meta-variables named in the language extension but also those for terminal and nonterminal forms that will be maintained from the base language. Otherwise, these meta-variables will be unbound in the extension language, leading to errors.

Nonterminals can be removed in an extended language by removing all of the productions of the nonterminal. New nonterminals can be added in an extended language by adding the productions of the new nonterminal. For instance, language `L10` removes the `Expr` nonterminal and adds `Effect`, `Pred`, and `Value` nonterminals.

```
(define-language L10
  (extends L9)
  (entry Program)
  (terminals
    (- (primitive (pr)))
    (+ (value-primitive (val-pr))
       (predicate-primitive (pred-pr))
       (effect-primitive (ef-pr))))
  (Program (prog)
    (- (letrec ([l* le*] ...) body))
    (+ (letrec ([l* le*] ...) vbody)))
  (Expr (e body)
    (- l
       x
       (quote c)
       (if e0 e1 e2)
       (begin e* ... e)
       (let ([x* e*] ...) body)
       (pr e* ...)
       (call e e* ...)))
  (Value (v vbody)
    (+ l
       x
       (quote c)
       (if p0 v1 v2)
       (begin e* ... v)
       (let ([x* v*] ...) vbody)
       (val-pr v* ...)
       (call v v* ...)))
  (Effect (e ebody)
    (+ (nop)
```

```
      (if p0 e1 e2)
      (begin e* ... e)
      (let ([x* v*] ...) ebody)
      (ef-pr v* ...)
      (call v v* ...)))
  (Pred (p pbody)
    (+ (true)
      (false)
      (if p0 p1 p2)
      (begin e* ... p)
      (let ([x* v*] ...) pbody)
      (pred-pr v* ...)))
  (LambdaExpr (le)
    (- (lambda (x* ...) body))
    (+ (lambda (x* ...) vbody))))
```

The L10 language also removes the `primitive` terminal and replaces it with the `value-primitive`, `predicate-primitive`, and `effect-primitive` terminals. These terminals are used for primitive calls in each of the three contexts specified in the language.

2.3.2.1. The `define-language` form. The `define-language` syntax has two related forms. The first form fully specifies a new language. The second form uses the `extends` clause to indicate that the language is an extension of an existing base language.

Both forms of `define-language` start with the same basic syntax:

`(define-language` *language-name clause* `...)`

where *clause* is an `extension` clause, an `entry` clause, a `terminals` clause, or a nonterminal clause.

Extension clause. The extension clause indicates that the new language is an extension of an existing language. This clause slightly changes the syntax of the `define-language` form and is described in Section 2.3.2.

Entry clause. The entry clause specifies which nonterminal is the starting point for this language. This information is used when generating passes to determine which nonterminal should be expected first by the pass. This default can be overridden in a pass definition, as described in Section 2.3.3.1. The entry clause has the following form:

`(entry` *nonterminal-name*`)`

where *nonterminal-name* corresponds to one of the nonterminals specified in this language. Only one entry clause can be specified in a language definition.

Terminals clause. The terminals clause specifies one or more terminals used by the language. For instance, in the `Lsrc` example language, the terminals clause specifies three terminal types: `uvar`, `primitive`, and `datum`. The terminals clause has the following form:

```
(terminals terminal-clause ...)
```

where *terminal-clause* has one of the following forms:

```
(terminal-name (meta-var ...))
(=> (terminal-name (meta-var ...)) prettifier)
(terminal-name (meta-var ...)) => prettifier
```

Here,

- *terminal-name* is the name of the terminal, and a corresponding *terminal-name*`?` predicate function exists to determine whether a Scheme object is of this type when checking the output of a pass,
- *meta-var* is the name of a meta-variable used for referring to this terminal type in language and pass definitions, and
- *prettifier* is a procedure expression of one argument used when the language unparser is called in "pretty" mode to produce a pretty, S-expression representation.

The final form is syntactic sugar for the form above it. When the *prettifier* is omitted, no processing is done on the terminal when the unparser runs.

Nonterminal clause. A nonterminal clause specifies the valid productions in a language. Each nonterminal clause has a name, a set of meta-variables, and a set of productions. A nonterminal clause has the following form:

```
(nonterminal-name (meta-var ...)
  production-clause
  ...)
```

where *nonterminal-name* is an identifier that names the nonterminal, *meta-var* is the name of a meta-variable used when referring to this nonterminal in language and pass definitions, and *production-clause* has one of the following forms:

```
terminal-meta-var
nonterminal-meta-var
production-s-expression
(keyword . production-s-expression)
```

Here,

- *terminal-meta-var* is a terminal meta-variable that is a stand-alone production for this nonterminal,
- *nonterminal-meta-var* is a nonterminal meta-variable that indicates that any form allowed by the specified nonterminal is also allowed by this nonterminal,
- *keyword* is an identifier that must be matched exactly when parsing an S-expression representation, language input pattern, or language output template, and
- *production-s-expression* is an S-expression that represents a pattern for production and has the following form:

meta-variable
`(maybe` *meta-variable*`)`
`(`*production-s-expression ellipsis*`)`
`(`*production-s-expression ellipsis production-s-expression* `... .` *production-s-expression*`)`
`(`*production-s-expression* `.` *production-s-expression*`)`
`()`

Here,

- *meta-variable* is any terminal or nonterminal meta-variable extended with an arbitrary number of digits, followed by an arbitrary combination of `*`, `?`, or `^` characters; for example, if the meta-variable is `e`, then `e1`, `e*`, `e?`, and `e4*?` are all valid meta-variable expressions;
- `(maybe` *meta-variable*`)` indicates that an element in the production is either of the type of the meta-variable or bottom (represented by `#f`); and
- *ellipsis* is the literal `...` and indicates that a list of the *production-s-expression* that proceeds it is expected.

Thus, a Scheme language form such as `let` can be represented as a language production as:

```
(let ([x* e*] ...) body* ... body)
```

where `let` is the *keyword*, `x*` is a meta-variable that indicates a list of variables, `e*` and `body*` are meta-variables that each indicate a list of expressions, and `body` is a meta-variable that indicates a single expression.

Using the `maybe` form, something similar to the named-let form could be represented as follows:

```
(let (maybe x) ([x* e*] ...) body* ... body)
```

although this would be slightly different from the normal named-let form, in that the non-named form would then need an explicit `#f` to indicate that no name was specified.

2.3.2.2. Extensions with the `define-language` form. A language defined as an extension of an existing language has a slightly modified syntax to indicate what should be added to or removed from the base language to create the new language. A compiler writer indicates that a language is an extension by using an extension clause.

Extension clause. The extension clause has the following form:

`(extends` *language-name*`)`

where *language-name* is the name of an already defined language. Only one extension clause can be specified in a language definition.

Entry clause. The entry clause does not change syntactically in an extended language. It can, however, name a nonterminal from the base language that is retained in the extended language.

Terminals clause. When a language derives from a base language, the `terminals` clause has the following form:

`(terminals` *extended-terminal-clause* `...)`

where *extended-terminal-clause* has one of the following forms:

`(+` *terminal-clause* `...)`
`(-` *terminal-clause* `...)`

where the *terminal-clause* uses the syntax for terminals specified in the non-extended version of the `terminals` form. The `+` form indicates terminals that should be added to the new language. The `-` form indicates terminals that should be removed from the list in the old language when producing the new language. Terminals not mentioned in a terminals clause will be copied unchanged into the new language. Note that adding and removing *meta-var*s from a terminal currently requires removing the terminal type and re-adding it. This can be done in the same step with a `terminals` clause, similar to the following:

```
(terminals
  (- (variable (x)))
  (+ (variable (x y))))
```

Nonterminal clause. When a language extends from a base language, a nonterminal clause has the following form:

```
(nonterminal-name (meta-var ...)
  extended-production-clause
  ...)
```

where *extended-production-clause* has one of the following forms:

```
(+ production-clause ...)
(- production-clause ...)
```

The `+` form indicates nonterminal productions that should be added to the nonterminal in the new language. The `-` form indicates nonterminal productions that should not be copied from the list of productions for this nonterminal in the base language when producing the new language. Productions not mentioned in a nonterminal clause will be copied unchanged into the nonterminal in the new language. If a nonterminal has all of its productions removed in a new language, the nonterminal will be dropped in the new language. Conversely, new nonterminals can be added by naming the new nonterminal and using the `+` form to specify the productions of the new nonterminal.

2.3.2.3. Products of `define-language`. The `define-language` form produces the following user-visible bindings:

- a language definition, bound to the specified *language-name*;
- an unparser (named `unparse-`*language-name*) that can be used to unparse a record-based representation back into an S-expression representation; and
- a set of predicates that can be used to identify a term of the language or a term from a specified nonterminal in the language.

It also produces the following internal bindings:

- a meta-parser that can be used by the `define-pass` macro to parse the patterns and templates used in passes and
- a set of record definitions that will be used to represent the language forms.

The `Lsrc` language, for example, will bind the identifier `Lsrc` to the language definition, produce an unparser named `unparse-Lsrc`, and create two predicates, `Lsrc?` and `Lsrc:Expr?`. The language definition is used when the *language-name* is specified as the base of a new language definition and in the definition of a pass.

The `define-parser` form can also be used to create a simple parser for parsing S-expressions into language forms as follows:

`(define-parser` *parser-name language-name*`)`

The parser does not support backtracking; thus, grammars must be specified, either by specifying a keyword or by having different length S-expressions so that the productions are unique.

For instance, the following language definition cannot be parsed because all four of the `set!` forms have the same keyword and are S-expressions of the same length:

```
(define-language Lunparsable
  (terminals
    (variable (x))
    (binop (binop))
    (integer-32 (int32))
    (integer-64 (int64)))
  (Program (prog)
    (begin stmt* ... stmt))
  (Statement (stmt)
    (set! x0 int64)
    (set! x0 x1)
    (set! x0 (binop x1 int32))
    (set! x0 (binop x1 x2))))
```

Instead, the `Statement` nonterminal must be broken into multiple nonterminals, as in the following language:

```
(define-language Lparsable
  (terminals
    (variable (x))
    (binop (binop))
    (integer-32 (int32))
    (integer-64 (int64)))
  (Program (prog)
    (begin stmt* ... stmt))
  (Statement (stmt)
    (set! x rhs))
  (Rhs (rhs)
    x
    int64
    (binop x arg))
  (Argument (arg)
    x
    int32))
```

2.3.3. Defining passes. Passes are used to specify transformations over languages defined by using `define-language`. Before going into the formal details of defining passes, we need to take a look at a

simple pass to convert an input program from the `Lsrc` intermediate language to the `L1` intermediate language. This pass removes the structured quoted datum by making the construction of the data explicit. To avoid constructing these constants more than once at run time, the pass also needs to lift the definitions of these datum to the outside of the program, binding them once and for all when the program begins running.

We define a pass called `convert-complex-datum` to accomplish this task, without using any of the catamorphism [**68**] or autogeneration features of the nanopass framework. Below, we can see how this feature helps eliminate boilerplate code.

```
(define-pass convert-complex-datum : Lsrc (x) -> L1 ()
  (definitions
    (define const-x* '())
    (define const-e* '())
    (with-output-language (L1 Expr)
      (define datum->expr
        (lambda (d)
          (cond
            [(pair? d) `(cons ,(datum->expr (car d)) ,(datum->expr (cdr d)))]
            [(vector? d)
             (let ([n (vector-length d)])
               (if (fxzero? n)
                   `(make-vector (quote 0))
                   (let ([t (unique-name 't)])
                     `(let ([,t (make-vector (quote ,n))])
                        (begin
                          ,(do ([i (fx- n 1) (fx- i 1)]
                                [ls '()
                                 (cons `(vector-set! ,t (quote ,i)
                                                     ,(datum->expr (vector-ref d i)))
                                       ls)])
                             ((< i 0) ls)) ...
                          ,t)))))]
            [else `(quote ,d)])))))
  (Expr : Expr (ir) -> Expr ()
    [,x x]
    [(quote ,d)
     (if (constant? d)
         `(quote ,d)
         (let ([t (unique-name 't)])
           (set! const-x* (cons t const-x*))
           (set! const-e* (cons (datum->expr d) const-e*))
           t))]
    [(if ,e0 ,e1 ,e2) `(if ,(Expr e0) ,(Expr e1) ,(Expr e2))]
    [(begin ,e* ... ,e) `(begin ,(map Expr e*) ... ,(Expr e))]
    [(lambda (,x* ...) ,body) `(lambda (,x* ...) ,(Expr body))]
    [(let ([,x* ,e*] ...) ,body)
     `(let ([,x* ,(map Expr e*)] ...) ,(Expr body))]
    [(letrec ([,x* ,e*] ...) ,body)
     `(letrec ([,x* ,(map Expr e*)] ...) ,(Expr body))]
```

```
    [(set! ,x ,e) `(set! ,x ,(Expr e))]
    [(,pr ,e* ...) `(,pr ,(map Expr e*) ...)]
    [(call ,e ,e* ...) `(call ,(Expr e) ,(map Expr e*) ...)])
  (let ([x (Expr x)])
    (if (null? const-x*)
        x
        `(let ([,const-x* ,const-e*] ...) ,x))))
```

The pass definition starts with a name (in this case, `convert-complex-datum`) and a signature. The signature starts with an input-language specifier (e.g. `Lsrc`), along with a list of formals. Here, there is just one formal, `x`, for the input-language term. The second part of the signature has an output-language specifier (in this case, `L1`), as well as a list of extra return values (in this case, empty).

Following the name and signature, this pass specifies definitions for `const-x*`, `const-e*`, and `datum->expr` in the `definitions` clause. The `definitions` clause can contain any Scheme expression valid in a definition context. The `const-x*` and `const-e*` variables are initialized to null and updated to contain the new binding and an expression to build the structured quoted datum, as each structured quoted datum is encountered. The `datum->expr` procedure recursively processes a structured quoted datum and produces the `L1` intermediate language code needed to construct it, using `cons` for building pairs and `make-vector` and `vector-set!` to build and fill vectors. The recursion in `datum->expr` terminates when a quoted constant is found. These definitions are scoped at the same level as the transformers in the pass.

Next, a transformer from the input nonterminal `Expr` to the output nonterminal `Expr` is defined. The transformer is named `Expr` and has a signature similar to that of the pass, with an input-language nonterminal and list of formals followed by the output-language nonterminal and list of extra-return-value expressions.

The transformer has a clause that processes each production of the `Expr` nonterminal. Each clause consists of an input pattern, an optional `guard` clause, and one or more expressions that specify zero or more return values based on the signature. The input pattern is derived from the S-expression productions specified in the input language. Each variable in the pattern is denoted by unquote (`,`). For instance, the clause for the `set!` production matches the pattern `(set! ,x ,e)`, binds `x` to the `uvar` specified by the `set!` and `e` to the `Expr` specified by the `set!`. The output-language expression is constructed using a quasiquoted template, in this case `` `(set! ,x ,(Expr e)) ``. Here, quasiquote, (`` ` ``), is rebound to a form that can construct language forms based on the template, and unquote

(,), is used to escape back into Scheme. The `,(Expr e)` thus puts the result of the recursive call of `Expr` into the output-language `(set! x e)` form.

Following the `Expr` transformer is the body of the pass, which calls `Expr` to transform the `Lsrc Expr` term into an `L1 Expr` term and wraps the result in a `let` expression if any structured quoted datum are found in the program that is being compiled.

In place of the explicit recursive calls to `Expr`, the compiler writer can use the catamorphism syntax to indicate the recurrence, as in the following version of the pass.

```
(define-pass convert-complex-datum : Lsrc (x) -> L1 ()
  (definitions
    (define const-x* '())
    (define const-e* '())
    (with-output-language (L1 Expr)
      (define datum->expr
        (lambda (d)
          (cond
            [(pair? d) `(cons ,(datum->expr (car d)) ,(datum->expr (cdr d)))]
            [(vector? d)
             (let ([n (vector-length d)])
               (if (fxzero? n)
                   `(make-vector (quote 0))
                   (let ([t (unique-name 't)])
                     `(let ([,t (make-vector (quote ,n))])
                        (begin
                          ,(map (lambda (i)
                                  `(vector-set! ,t (quote ,i)
                                     ,(datum->expr (vector-ref d i))))
                             (iota n))
                          ...
                          ,t)))))]
            [else `(quote ,d)])))))
  (Expr : Expr (ir) -> Expr ()
    [,x x]
    [(quote ,d)
     (if (constant? d)
         `(quote ,d)
         (let ([t (unique-name 't)])
           (set! const-x* (cons t const-x*))
           (set! const-e* (cons (datum->expr d) const-e*))
           t))]
    [(if ,[e0] ,[e1] ,[e2]) `(if ,e0 ,e1 ,e2)]
    [(begin ,[e*] ... ,[e]) `(begin ,e* ... ,e)]
    [(lambda (,x* ...) ,[body]) `(lambda (,x* ...) ,body)]
    [(let ([,x* ,[e*]] ...) ,[body]) `(let ([,x* ,e*] ...) ,body)]
    [(letrec ([,x* ,[e*]] ...) ,[body]) `(letrec ([,x* ,e*] ...) ,body)]
    [(set! ,x ,[e]) `(set! ,x ,e)]
    [(,pr ,[e*] ...) `(,pr ,e* ...)]
    [(call ,[e] ,[e*] ...) `(call ,e ,e* ...)]
    [else (errorf who "invalid Expr form ~s" ir)])
```

```
(let ([x (Expr x)])
  (if (null? const-x*)
      x
      `(let ([,const-x* ,const-e*] ...) ,x))))
```

Here, the square brackets that wrap the unquoted variable expression in a pattern indicate that a catamorphism should be applied. For instance, in the `set!` clause, the `,e` from the previous pass becomes `,[e]`. When the catamorphism is included on an element that is followed by an ellipsis, `map` is used to process the elements of the list and to construct the output list.

Using catamorphisms helps to make the pass more succinct, but there is still boilerplate code in the pass that the framework can fill in for the compiler writer. Several clauses simply match the input-language production and generate a matching output-language production (modulo the catamorphisms for nested `Expr` forms). Because the input and output languages are defined, the `define-pass` macro can automatically generate these clauses. Thus, the same functionality can be expressed as follows:

```
(define-pass convert-complex-datum : Lsrc (x) -> L1 ()
  (definitions
    (define const-x* '())
    (define const-e* '())
    (with-output-language (L1 Expr)
      (define datum->expr
        (lambda (d)
          (cond
            [(pair? d) `(cons ,(datum->expr (car d)) ,(datum->expr (cdr d)))]
            [(vector? d)
             (let ([n (vector-length d)])
               (if (fxzero? n)
                   `(make-vector (quote 0))
                   (let ([t (unique-name 't)])
                     `(let ([,t (make-vector (quote ,n))])
                        (begin
                          ,(do ([i (fx- n 1) (fx- i 1)]
                                [ls '()
                                 (cons `(vector-set! ,t (quote ,i)
                                                    ,(datum->expr (vector-ref d i)))
                                       ls)])
                               ((< i 0) ls)) ...
                          ,t)))))]
            [else `(quote ,d)])))))
  (Expr : Expr (ir) -> Expr ()
    [(quote ,d)
     (guard (not (constant? d)))
     (let ([t (unique-name 't)])
       (set! const-x* (cons t const-x*))
       (set! const-e* (cons (datum->expr d) const-e*))
       t)])
```

```
  (let ([x (Expr x)])
    (if (null? const-x*)
        x
        `(let ([,const-x* ,const-e*] ...) ,x))))
```

In this version of the pass, only the `quote` form where the datum is not already a constant is explicitly processed. The `define-pass` form automatically generates the other clauses. Although all three versions of this pass perform the same task, the final form is the closest to the initial intention of changing just the complex quoted datum.

In addition to `define-pass` autogenerating the clauses of a transformer, `define-pass` can also autogenerate the transformers for nonterminals that must be traversed but are otherwise unchanged in a pass. For instance, one of the passes in the class compiler removes complex expressions from the right-hand side of the `set!` form. At this point in the compiler, the language has several nonterminals:

```
(define-language L14 (entry Program)
  (terminals
    (relative-operator (relop))
    (binary-operator (binop))
    (immediate (imm))
    (label (l))
    (uvar (x)))
  (Pred (p pbody)
    (true)
    (false)
    (relop t0 t1)
    (if p0 p1 p2)
    (begin e* ... p))
  (Effect (e ebody)
    (nop)
    (set! x v)
    (mset! t0 t1 t2)
    (call t t* ...)
    (if p0 e1 e2)
    (begin e* ... e))
  (Value (v vbody)
    t
    (alloc t)
    (mref t0 t1)
    (binop t0 t1)
    (call t t* ...)
    (if p0 v1 v2)
    (begin e* ... v))
  (Program (prog)
    (letrec ([l* le*] ...) lbody))
  (LambdaExpr (le)
    (lambda (x* ...) lbody))
  (LocalsBody (lbody)
```

```
    (locals (x* ...) vbody))
  (Triv (t)
    x
    l
    imm))
```

The pass, however, is only interested in the `set!` form and the `Value` form in the right-hand-side position of the `set!` form. Relying on the autogeneration of transformers, this pass can be written as:

```
(define-pass flatten-set! : L14 (x) -> L15 ()
  (trivialize : Value (v x) -> Effect ()
    [(if ,[p0] ,[e1] ,[e2]) `(if ,p0 ,e1 ,e2)]
    [(begin ,[e*] ... ,[e]) `(begin ,e* ... ,e)]
    [(,binop ,t0 ,t1) `(set! ,x (,binop ,t0 ,t1))]
    [(call ,t ,t* ...) `(set! ,x (call ,t ,t* ...))]
    [(mref ,t0 ,t1) `(set! ,x (mref ,t0 ,t1))]
    [(alloc ,t) `(set! ,x (alloc ,t))]
    [,t `(set! ,x ,t)]
    [else (errorf who "unrecognized Value ~s" v)])
  (Effect : Effect (x) -> Effect ()
    [(set! ,x ,v) (trivialize v x)]))
```

Here, the `Effect` transformer has just one clause for matching the `set!` form. The `trivialize` transformer is called to produce the final `Effect` form. The `trivialize` transformer, in turn, pushes the `set!` form into the `if` and `begin` forms and processes the contents of these forms, which produces a final `Effect` form. The catamorphisms in the `if` and `begin` clauses automatically recur to `trivialize` based on the type signature of `Value -> Effect` and pass along the `set!` form from the left-hand side `x`. The `define-pass` macro autogenerates transformers for `Program`, `LambdaExpr`, `LocalsBody`, `Triv`, `Value`, and `Pred` that recur through the input-language forms and produce the output-language forms.

It is sometimes necessary to pass more information than just the language term to a transformer. The transformer syntax allows extra formals to be named to support passing this information. For example, in the pass from the class compiler that converts the `closures` form into explicit calls to procedure primitives, the closure pointer, `cp`, and the list of free variables, `free*`, are passed to the `ClosureBody`, `uvar`, and `Expr` transformers.

```
(define-pass introduce-procedure-primitives : L7 (x) -> L8 ()
  (definitions
    (define index-of
      (lambda (x ls)
        (let f ([ls ls] [i 0])
          (and (not (null? ls))
```

```
            (if (eq? x (car ls))
                i
                (f (cdr ls) (+ i 1))))))))
  (ClosureBody : ClosureBody (x cp free*) -> Expr ()
    [(closures ([,x* ,l* ,[e**] ...] ...) ,[body])
     (let ([len* (map length e**)])
       `(let ([,x* (make-procedure ,l* (quote ,len*))] ...)
          (begin
            ,(fold-left
               (lambda (ls lhs* i* free*)
                 (fold-left
                   (lambda (ls lhs i free)
                     (cons `(procedure-set! ,lhs (quote ,i) ,free) ls))
                   ls lhs* i* free*))
               '() (map make-list len* x*) (map iota len*) e**)
            ...
            ,body)))])
  (uvar : uvar (uvar cp free*) -> Expr ()
    (cond
      [(index-of uvar free*) => (lambda (i) `(procedure-ref ,cp (quote ,i)))]
      [else uvar]))
  (Expr : Expr (x cp free*) -> Expr ()
    [(call ,l ,[e*] ...) `(call ,l ,e* ...)]
    [(call ,[e] ,[e*] ...) `(call (procedure-code ,e) ,e* ...)])
  (LambdaExpr : LambdaExpr (x) -> LambdaExpr ()
    [(lambda (,x* ...) (bind-free (,x ,x1* ...) ,[body x x1* -> body]))
     `(lambda (,x* ...) ,body)])
  (Expr x #f '()))
```

The catamorphism and clause autogeneration facilities are also aware of the extra formals expected by transformers. In a catamorphism, this means that extra arguments need not be specified in the catamorphism, if the formals are available in the transformer. For instance, in both the `ClosureBody` and `Expr` transformers, the catamorphism specifies only the binding of the output `Expr` form, and `define-pass` matches the name of the formal to the transformer with the expected argument. In the `LambdaExpr` transformer, the extra arguments need to be specified, both because they are not available as a formal of the transformer and because the values change at the `LambdaExpr` boundary. Autogenerated clauses in `Expr` also call the `uvar` and `Expr` transformers with the extra arguments from the formals.

In some cases, it is useful to specify a default value for an extra argument. For example, in the `convert-assignments` pass, the `set*` argument can be defaulted to null, as the list of assigned variables is built up as `let` and `lambda` forms are encountered that have assigned variables.

```
(define-pass convert-assignments : L3 (x) -> L4 ()
  (definitions
    (define replace
      (lambda (set* t*)
```

```
      (lambda (x)
        (let f ([set* set*] [t* t*])
          (if (null? set*)
              x
              (if (eq? (car set*) x)
                  (car t*)
                  (f (cdr set*) (cdr t*))))))))))
  (Expr : Expr (x [set* '()]) -> Expr ()
    [,x (if (memq x set*) `(car ,x) x)]
    [(set! ,x ,[e]) `(set-car! ,x ,e)]
    [(let ([,x* ,[e*]] ...) (assigned () ,[body]))
     `(let ([,x* ,e*] ...) ,body)]
    [(let ([,x* ,[e*]] ...) (assigned (,x1* ...) ,body))
      (let ([t* (map unique-name x1*)])
        (let ([x* (map (replace x1* t*) x*)])
          `(let ([,x* ,e*] ...)
             (let ([,x1* ,(map (lambda (t) `(cons ,t (void))) t*)] ...)
               ,(Expr body (append x1* set*))))))])
  (LambdaExpr : LambdaExpr (x [set* '()]) -> LambdaExpr ()
    [(lambda (,x* ...) (assigned () ,[body])) `(lambda (,x* ...) ,body)]
    [(lambda (,x* ...) (assigned (,x1* ...) ,body))
      (let ([t* (map unique-name x1*)])
        (let ([x* (map (replace x1* t*) x*)])
          `(lambda (,x* ...)
             (let ([,x1* ,(map (lambda (t) `(cons ,t (void))) t*)] ...)
               ,(Expr body (append x1* set*))))))]))
```

The catamorphisms and recursion on sub-nonterminals in the autogenerated clauses for the `Expr` and `LambdaExpr` transformers will pass along the current value of `set*`, as the `set*` formal is available in these transformers. The `lambda` and `let` clauses that contain a non-empty assigned variable list add their assigned variables to the `set*` list. When `define-pass` autogenerates the body for the `convert-assignments` pass, it will use the default value for `set*` from the `Expr` transformer in the initial call to the `Expr` transformer.

Transformers can also be written that handle terminals instead of nonterminals. Because terminals have no structure, the body of such transformers is simply a Scheme expression. For example, in the pass that replaces variables with their locations after register assignment, the `uvar` terminal is replaced by the `location`.

```
(define-pass finalize-locations : L23 (x) -> L24 ()
  (uvar-reg-fv : uvar-reg-fv (x env) -> location ()
    (cond [(and (uvar? x) (assq x env)) => cdr] [else x]))
  (Triv : Triv (x env) -> Triv ())
  (Rhs : Rhs (x env) -> Rhs ())
  (Pred : Pred (x env) -> Pred ())
  (Effect : Effect (x env) -> Effect ())
  (Value : Value (x env) -> Value ())
```

```
  (LocalsBody : LocalsBody (x) -> Value ()
    [(finished ([,x* ,loc*] ...) ,vbody) (Value vbody (map cons x* loc*))]))
```

The two interesting parts of this pass are the `LocalsBody` transformer that creates the environment that maps variables to locations and the `uvar-reg-fv` transformer that replaces variables with the appropriate location. In this pass, transformers cannot be autogenerated because extra arguments are needed, and the nanopass framework only autogenerates transformers without extra arguments or return values. The autogeneration is limited to help reign in some of the unpredictable behavior that can result from autogenerated transformers.

Passes can also be written that do not take a language form but that produce a language form. The initial parser for the class compiler is a good example of this. It expects an S-expression that conforms to an input grammar for the subset of Scheme used by the class compiler.

```
(define-pass parse-scheme : * (x) -> Lsrc ()
  (definitions —)
  (Expr : * (x env) -> Expr ()
    (cond
      [(constant? x) `(quote ,x)]
      [(symbol? x)
       (let ([uv (lookup x env)])
         (if (uvar? uv) uv (uv env x)))]
      [(and (pair? x) (let ([id (car x)]) (and (symbol? id) id))) =>
       (lambda (id)
         (let ([uv (lookup id env)])
           (if (uvar? uv) (Application env id (cdr x)) (uv env (cdr x)))))]
      [(pair? x)
       (let ([id (car x)])
         (cond
           [(and (symbol? id) (let ([uv (lookup id env)]) (and (procedure? uv)) uv)) =>
            (lambda (uv) (uv env (cdr x)))]
           [else (Application env id (cdr x))]))]
      [else (errorf who "invalid Expr ~s" x)]))
  (Application : * (env fun arg*) -> Expr ()
    (cond
      [(list? arg*) `(call ,(Expr fun env) ,(Expr* arg* env) ...)]
      [else (errorf who "invalid Expr ~s" (cons fun arg*))]))
  (let ()
    (define build-begin
      (lambda (env e*)
        (let loop ([e (car e*)] [e* (cdr e*)] [re* '()])
          (if (null? e*)
              (let ([e (Expr e env)])
                (if (null? re*) e `(begin ,(reverse re*) ... ,e)))
              (loop (car e*) (cdr e*) (cons (Expr e env) re*))))))
    (define (do-primitive prim n)
      (lambda (env arg*)
        (unless (= (length arg*) n) (errorf who "invalid Expr ~s" (cons prim arg*)))
        `(,prim ,(Expr* arg* env) ...)))
```

```
(define initial-environment empty-env)
(define (add-initial-binding! id action)
  (set! initial-environment (extend initial-environment id action)))
(define-syntax add-prim-binding!
  (syntax-rules ()
    [(_ prim)
      (add-initial-binding! 'prim (do-primitive 'prim (primitive->arity 'prim)))]))
(define-syntax add-prim-binding*!
  (syntax-rules ()
    [(_ prim0 prim1 ...)
      (begin (add-prim-binding! prim0) (add-prim-binding! prim1) ...)]))
; keywords in the language
(add-initial-binding! 'quote
  (lambda (env e*)
    (unless (and (list? e*) (fx= (length e*) 1))
      (errorf who "invalid Expr ~s" (cons 'quote e*)))
    (let ([d (car e*)])
      (unless (datum? d) (errorf who "invalid datum ~s" d))
      `(quote ,d))))
—
; macro style primitives
(add-initial-binding! 'and
  (lambda (env e*)
    (unless (list? e*) (errorf who "invalid Expr ~s" (cons 'and e*)))
    (if (null? e*)
        `(quote #t)
        (let f ([e* (Expr* e* env)])
          (let ([e (car e*)] [e* (cdr e*)])
            (if (null? e*)
                e
                `(if ,e ,(f e*) (quote #f))))))))
—
(add-prim-binding*! procedure? + - * car cdr cons make-vector vector-length
  vector-ref void < <= = >= > boolean? eq? fixnum? null? pair? vector?
  set-car! set-cdr! vector-set!)
(Expr x initial-environment)))
```

The `parse-scheme` pass is structured similarly to a simple expander with keywords, macros,[2] and primitives. It also performs syntax checking to ensure that the input grammar conforms to the expected input grammar. Finally, it produces an `Lsrc` language term that represents the Scheme program to be compiled.

In the pass syntax, the `*` in place of the input-language name indicates that no input-language term should be expected. The `Expr` and `Application` transformers do not have pattern matching clauses, as the input could be of any form. The quasiquote is, however, rebound because an output language is specified.

[2]In this case, just the fixed `and` macro and `or` macro are defined.

It can also be useful to create passes without an output language. The final pass of the class compiler is the code generator that emits x86_64 assembler code.

```
(define-pass generate-x86_64 : L28 (x) -> * ()
  (definitions
    (define prim->opcode
      (lambda (prim)
        (cdr (assq prim
               '((+ . addq) (- . subq) (* . imulq)
                 (logand . andq) (logor . orq) (sra . sarq))))))
    (define relop->opcode
      (lambda (relop not?)
        (cdr (assq relop
               (if not?
                   '((= . jne) (< . jge) (<= . jg) (> . jle) (>= . jl))
                   '((= . je) (< . jl) (<= . jle) (> . jg) (>= . jge))))))))
  (Code : Code (x) -> * ()
    [(label ,l) (emit-label l)]
    [(jump ,t) (emit-jump 'jmp t)]
    [(set! ,x ,l) (emit 'leaq l x)]
    [(set! ,x0 (,binop ,x1 ,t2))
     (emit (prim->opcode binop) t2 x1)]
    [(set! ,x0 ,t1) (emit 'movq t1 x0)]
    [(set! ,x0 ,rhs) (errorf who "unrecognized set! expression ~s" x)]
    [(if (not (,relop ,t0 ,t1)) (,l))
     (emit 'cmpq t1 t0)
     (emit-jump (relop->opcode relop #t) l)]
    [(if (,relop ,t0 ,t1) (,l))
     (emit 'cmpq t1 t0)
     (emit-jump (relop->opcode relop #f) l)]
    [(if ,p0 (,l))
     (errorf who "unrecognized if expression ~s" x)])
  (Program : Program (x) -> * ()
    [(code ,c* ...) (emit-program (for-each Code c*))]))
```

Again, a `*` is used to indicate that there is no language form in this case for the output language. The assembly code is printed to the standard output port. Thus, there is no need for any return value from this pass.

Passes can also return a value that is not a language form. For instance, the `everybody-home?` predicate used to determine when register allocation is complete can be written as a pass that returns a boolean value.

```
(define-pass everybody-home? : L22 (x) -> * (bool)
  (home? : LocalsBody (x) -> * (bool)
    [(locals (,x* ...) ,ulbody) #f]
    [(finished ([,x* ,loc*] ...) ,vbody) #t])
  (Program : Program (x) -> * (bool)
    [(letrec ([,l* (lambda () ,lbody*)] ...) ,lbody)
```

```
    (andmap home? (cons lbody lbody*))]
   [(letrec ([,l* ,le*] ...) ,lbody)
    (errorf who "program with unexpected shape ~s" x)])
 (Program x))
```

Here, the extra return value is indicated as `bool`. The `bool` here is used to indicate to `define-pass` that an extra value is being returned. Any expression can be used in this position. In this case, the `bool` identifier will simply be an unbound variable if it is ever manifested. It is not manifested in this case, however, because the body is explicitly specified; thus, no code will be autogenerated for the body of the pass.

2.3.3.1. The `define-pass` syntactic form. The `define-pass` form has the following syntax.

```
(define-pass name : lang-specifier (fml ...) -> lang-specifier (extra-return-val-expr ...)
  definitions-clause
  transformer-clause ...
  body-expr ...)
```

where *name* is an identifier to use as the name for the procedure definition. The *lang-specifier* has one of the following forms:

```
*
lang-name
(lang-name  nonterminal-name)
```

where

- *lang-name* refers to a language defined with the `define-language` form, and
- *nonterminal-name* refers to a nonterminal named within the language definition.

When the `*` form is used as the input *lang-specifier*, it indicates that the pass does not expect an input-language term. When there is no input language, the transformers within the pass do not have clauses with pattern matches because, without an input language, the `define-pass` macro does not know what the structure of the input term will be. When the `*` form is used as the output *lang-specifier*, it indicates that the pass does not produce an output-language term and should not be checked. When there is no output language, the transformers within the pass do not bind `quasiquote`, and there are no templates on the right-hand side of the transformer matches. It is possible to use the `*` specifier for both the input and output *lang-specifier*. This effectively turns the pass, and the transformers contained within it, into an ordinary Scheme function.

When the *lang-name* form is used as the input *lang-specifier*, it indicates that the pass expects an input-language term that is one of the productions from the entry nonterminal. When the *lang-name* form is used as the output *lang-specifier*, it indicates that the pass expects that an output-language term will be produced and checked to be one of the records that represents a production of the entry nonterminal.

When the (*lang-name nonterminal-name*) form is used as the input-language specifier, it indicates that the input-language term will be a production from the specified nonterminal in the specified input language. When the (*lang-name nonterminal-name*) form is used as the output-language specifier, it indicates that the pass will produce an output production from the specified nonterminal of the specified output language.

The *fml* is a Scheme identifier, and if the input *lang-specifier* is not `*`, the first *fml* refers to the input-language term.

The *extra-return-val-expr* is any valid Scheme expression that is valid in value context. These expressions are scoped within the binding of the identifiers named as *fml*s.

The optional *definitions-clause* has the following form:

`(definitions` *scheme-definition* `...)`

where *scheme-definition* is any Scheme expression that can be used in definition context. Definitions in the *definitions-clause* are in the same lexical scope as the transformers, which means that procedures and macros defined in the *definitions-clause* can refer to any transformer named in a *transformer-clause*.

The *definitions-clause* is followed by zero or more *transformer-clauses* of the following form:

`(`*name* `:` *nt-specifier* `(`*fml-expr* `...) ->` *nt-specifier* `(`*extra-return-val-expr* `...)`
  definitions-clause`?`
  transformer-body`)`

where *name* is a Scheme identifier that can be used to refer to the transformer within the pass. The input *nt-specifier* is one of the following two forms:

`*`
nonterminal-name

When the `*` form is used as the input nonterminal, it indicates that no input nonterminal form is expected and that the body of the *transformer-body* will not contain pattern matching clauses.

When the `*` form is used as the output nonterminal, `quasiquote` will not be rebound, and no output-language templates are available. When both the input and output *nt-specifier* are `*`, the transformer is effectively an ordinary Scheme procedure.

The *fml-expr* has one of the following two forms:

fml
`[`*fml default-val-expr*`]`

where *fml* is a Scheme identifier and *default-val-expr* is a Scheme expression. The *default-val-expr* is used when an argument is not specified in a catamorphism or when a matching `fml` is not available in the calling transformer. All arguments must be explicitly provided when the transformer is called as an ordinary Scheme procedure. Using the catamorphism syntax, the arguments can be explicitly supplied, using the syntax discussed on page 40. It can also be specified implicitly. Arguments are filled in implicitly in catamorphisms that do not explicitly provide the arguments and in autogenerated clauses when the nonterminal elements of a production are processed. These implicitly supplied formals are handled by looking for a formal in the calling transformer that has the same name as the formal expected by the target transformer. If no matching formal is found, and the target transformer specifies a default value, the default value will be used in the call; otherwise, another target transformer must be found, a new transformer must be autogenerated, or an exception must be raised to indicate that no transformer was found and none can be autogenerated.

The *extra-return-val-expr* can be any Scheme expression. These expressions are scoped within the *fml*s bound by the transformer. This allows an input formal to be returned as an extra return value, implicitly in the autogenerated clauses. This can be useful for threading values through a transformer.

The optional *definitions-clause* can include any Scheme expression that can be placed in a definition context. These definitions are scoped within the transformer. When an output nonterminal is specified, the `quasiquote` is also bound within the body of the `definitions` clause to allow language term templates to be included in the body of the definitions.

When the input *nt-specifier* is not `*`, the *transformer-body* has one of the following forms:

`[`*pattern guard-clause body** `...` *body*`]`
`[`*pattern body** `...` *body*`]`
`[else` *body** `...` *body*`]`

where the `else` clause must be the last one listed in a transformer and prevents autogeneration of missing clauses (because the `else` clause is used in place of the autogenerated clauses). The *pattern* is an S-expression pattern, based on the S-expression productions used in the language definition. Patterns can be arbitrarily nested. Variables bound by the pattern are preceded by an `unquote` and are named based on the meta-variables named in the language definition. The variable name can be used to restrict the pattern by using a meta-variable that is more specific than the one specified in the language definition. The *pattern* can also contain catamorphisms that have one of the following forms:

```
[Proc-expr : input-fml arg ... -> output-fml extra-rv-fml ...]
[Transformer-name : output-fml extra-rv-fml ...]
[input-fml arg ... -> output-fml extra-rv-fml ...]
[output-fml extra-rv-fml ...]
```

In the first form, the *Proc-expr* is an explicitly specified procedure expression (which may be a *Transformer-name*), the *input-fml* and all arguments to the procedure are explicitly specified, and the results of calling the *Proc-expr* are bound by the *output-fml* and *extra-rv-fml*s. In the second form, the *Transformer-name* is an identifier that refers to a transformer named in this pass. The `define-pass` macro determines, based on the signature of the transformer referred to by the *Transformer-name*, what arguments should be supplied to the transformer. In the last two forms, the transformer is determined automatically. In the third form, the nonterminal type associated with the *input-fml*, the *arg*s, the output nonterminal type based on the *output-fml*, and the *extra-rv-fml*s are used to determine the transformer to call. In the final form, the nonterminal type for the field within the production, along with the formals to the calling transformer, the output nonterminal type based on the *output-fml*, and the *extra-rv-fml*s are used to determine the transformer to call. In the two forms where the transformer is not explicitly named, a new transformer can be autogenerated when no *arg*s and no *extra-rv-fml*s are specified. This limitation is in place to avoid creating a transformer with extra formals whose use is unspecified and extra return values with potentially dubious return-value expressions.

The *input-fml* is a Scheme identifier with a name based on the meta-variables named in the input-language definition. The specification of a more restrictive meta-variable name can be used to further restrict the pattern. The *output-fml* is a Scheme identifier with a name based on the meta-variables named in the output-language definition. The *extra-rv-fml* is a Scheme identifier. The *input-fml*s named in the fields of a pattern must be unique. The *output-fml*s and *extra-rv-fml*s must also

be unique, although they can overlap with the *input-fml*s that are shadowed in the body by the *output-fml* or *extra-rv-fml* with the same name.

Only the *input-fml*s are visible within the optional *guard-clause*. This is because the *guard-clause* is evaluated before the catamorphisms recur on the fields of a production. The *guard-clause* has the following form:

`(guard` *guard-expr* `...)`

where *guard-expr* is a Scheme expression. The *guard-clause* has the same semantics as `and`.

The *body** and *body* are any Scheme expression. When the output *nt-specifier* is not `*`, `quasiquote` is rebound to a macro that interprets `quasiquote` expressions as templates for productions in the output nonterminal. Additionally, `in-context` is a macro that can be used to rebind `quasiquote` to a different nonterminal. Templates are specified as S-expressions based on the productions specified by the output language. In templates, `unquote` is used to indicate that the expression in the `unquote` should be used to fill in the given field of the production. Within an `unquote` expression, `quasiquote` is rebound to the appropriate nonterminal based on the expected type of the field in the production. If the template includes items that are not `unquote`d where a field value is expected, the expression found there is automatically quoted. This allows self-evaluating items such as symbols, booleans, and numbers to be more easily specified in templates. A list of items can be specified in a field that expects a list, using an ellipsis.

Although the syntax of a language production is specified as an S-expression, the record representation used for the language term separates each variable specified into a separate field. This means that the template syntax expects a separate value or list of values for each field in the record. For instance, in the `(letrec ([x* e*] ...) body)` production, it is not possible to have a template of the form `(letrec (,bindings ...) ,body)` because the nanopass framework will not attempt to break up the `bindings` list into its `x*` and `e*` component parts. The same can be accomplished by the template `(letrec ([,(map car bindings) ,(map cadr bindings)] ...) ,body)`. It is possible that the nanopass framework could be extended to perform the task of splitting up the `binding*` list automatically, but it is not done currently, partially to avoid hiding the cost of deconstructing the `binding*` list and constructing the `x*` and `e*` lists.

The `in-context` expression within the body has the following form:

`(in-context` *nonterminal-name body** `...` *body*`)`

The `in-context` form rebinds the `quasiquote` to allow productions from the named nonterminal to be constructed in a context where they are not otherwise expected.

2.3.4. Constructing language forms outside of a pass. In addition to creating language forms using a parser defined with `define-parser` or through a pass defined with `define-pass`, language forms can also be created using the `with-output-language` form. The `with-output-language` form binds the `in-context` transformer for the language specified and, if a nonterminal is also specified, binds the `quasiquote` form. This allows the same template syntax used in the body of a transformer to be used outside of the context of a pass. In a commercial compiler, such as Chez Scheme, it is often convenient to use functional abstraction to centralize the creation of a language term.

For instance, in the `convert-complex-datum` pass discussed in Section 2.3.3, the `datum->expr` procedure has a `with-output-language` wrapped around the body of the function. This is done so that primitive calls to `cons`, `make-vector`, and `vector-set!` can be constructed, along with quoted constants, `let`, and `begin` forms. Looking more closely at the `datum->expr` procedure, we can see that the `with-output-language` syntax specifies a language and a nonterminal to be used.

```
(define datum->expr
  (with-output-language (L1 Expr)
    (lambda (d)
      (cond
        [(pair? d) `(cons ,(datum->expr (car d)) ,(datum->expr (cdr d)))]
        [(vector? d)
         (let ([n (vector-length d)])
           (if (fxzero? n)
               `(make-vector (quote 0))
               (let ([t (unique-name 't)])
                 `(let ([,t (make-vector (quote ,n))])
                    (begin
                      ,(do ([i (fx- n 1) (fx- i 1)]
                            [ls '()
                             (cons `(vector-set! ,t (quote ,i)
                                                 ,(datum->expr (vector-ref d i)))
                                   ls)])
                         ((< i 0) ls)) ...
                      ,t)))))]
        [else `(quote ,d)]))))
```

This rebinds both the `quasiquote` keyword and the `in-context` keyword.

The `with-output-language` form has one of the following forms:

```
(with-output-language lang-name expr* ... expr)
(with-output-language (lang-name nonterminal-name) expr* ... expr)
```

In the first form, the `in-context` form is bound and can be used to specify a *nonterminal-name*, as described at the end of Section 2.3.3. In the second form, both `in-context` and `quasiquote` are bound. The `quasiquote` form is bound in the context of the specified *nonterminal-name*, and templates can be defined just as they are on the right-hand side of a transformer clause.

The `with-output-language` form is a splicing form, similar to `begin` or `let-syntax`, allowing multiple definitions or expressions that are all at the same scoping level as the `with-output-language` form to be contained within the form. This is convenient when writing a set of definitions that all construct some piece of a language term from the same nonterminal. This flexibility means that the `with-output-language` form cannot be defined as syntactic sugar for the `define-pass` form.

2.3.5. Matching language forms outside of a pass. In addition to the `define-pass` form, it is possible to match a language term using the `nanopass-case` form. This can be useful when creating functional abstractions, such as predicates that ask a question based on matching a language form. For instance, suppose we write a `constant-form?` predicate for the `L0` language as follows:

```
(define constant-form?
  (lambda (x)
    (nanopass-case (L0 Expr) x
      [,c #t]
      [(quote ,d) #t]
      [else #f])))
```

The `nanopass-case` form has the following syntax:

```
(nanopass-case (lang-name nonterminal-name) expr
  matching-clause ...)
```

where *matching-clause* has one of the following forms:

```
[pattern guard-clause expr* ... expr]
[pattern expr* ... expr]
[else expr* ... expr]
```

where the *pattern* and *guard-clause* forms have the same syntax as in the *transformer-body* of a pass.

Similar to `with-output-language`, `nanopass-case` provides a more succinct syntax for matching a language form than does the general `define-pass` form. Unlike the `with-output-language` form, however, the `nanopass-case` form can be implemented in terms of the `define-pass` form. For example, the `constant-form?` predicate also could have been written as:

```
(define-pass constant-form? : (L0 Expr) (ir) -> * (bool)
  (Expr : Expr (ir) -> * (bool)
    [,c #t]
    [(quote ,d) #t]
    [else #f])
  (Expr ir))
```

This is, in fact, how the `nanopass-case` macro is implemented.

2.3.6. Working with languages.

2.3.6.1. Displaying languages. The full definition of a language can be printed by supplying the language name to the `language->s-expression` form. This can be helpful when working with extended languages, such as in the case of `L1`:

```
(language->s-expression L1)
```

which returns:

```
(define-language L1
  (entry Expr)
  (terminals
    (constant (c))
    (primitive (pr))
    (uvar (x)))
  (Expr (e body)
    (quote c)
    x
    (if e0 e1 e2)
    (begin e* ... e)
    (lambda (x* ...) body)
    (let ([x* e*] ...) body)
    (letrec ([x* e*] ...) body)
    (set! x e)
    (pr e* ...)
    (call e e* ...)))
```

2.3.6.2. Differencing languages. The extension form can also be derived between any two languages by using the `diff-languages` form. For instance, we can get the differences between the `Lsrc` and `L1` language (giving us back the extension) with:

```
(diff-languages Lsrc L1)
```

which returns:

```
(define-language L1
  (extends Lsrc)
  (entry Expr)
  (terminals
    (- (datum (d)))
    (+ (constant (c))))
  (Expr (body e)
    (- (quote d))
    (+ (quote c))))
```

2.3.6.3. Viewing the expansion of passes and transformers. The `define-pass` form autogenerates both transformers and clauses within transformers. In simple passes, these are generally straightforward to reason about, but in more complex passes, particularly those that make use of different arguments for different transformers or include extra return values, it can become more difficult to determine what code will be generated. In particular, the experience of developing a full commercial compiler has shown that the `define-pass` form can autogenerate transformers that shadow those defined by the compiler writer. To help the compiler writer determine what code is being generated, there is a variation of the `define-pass` form, called `echo-define-pass`, that will echo the expansion of `define-pass`.

For instance, we can echo the `convert-complex-datum` pass to get the following:

```
(echo-define-pass convert-complex-datum : Lsrc (x) -> L1 ()
  (definitions
    (define const-x* '())
    (define const-e* '())
    (define datum->expr
      (with-output-language (L1 Expr)
        —)))
  (Expr : Expr (ir) -> Expr ()
    [(quote ,d)
     (guard (not (constant? d)))
     (let ([t (unique-name 't)])
       (set! const-x* (cons t const-x*))
       (set! const-e* (cons (datum->expr d) const-e*))
       t)])
  (let ([x (Expr x)])
    (if (null? const-x*)
        x
        `(let ([,const-x* ,const-e*] ...) ,x))))
 ⇒

(define convert-complex-datum
  (lambda (x)
    (define who 'convert-complex-datum)
    (define-nanopass-record)
    (define const-x* '())
    (define const-e* '())
```

```
(define datum->expr
  (with-output-language (L1 Expr)
    —))
(define Expr
  (lambda (ir)
    (let ([g0.14 ir])
      (let-syntax ([quasiquote '#<procedure>]
                   [in-context '#<procedure>])
        (begin
          (let ([rhs.15 (lambda (d)
                          (let ([t (unique-name 't)])
                            (set! const-x* (cons t const-x*))
                            (set! const-e*
                              (cons (datum->expr d) const-e*))
                            t))])
            (cond
              [(uvar? g0.14) g0.14]
              [else
               (let ([tag (nanopass-record-tag g0.14)])
                 (cond
                   [(eqv? tag 1)
                    (let* ([d (Lsrc:quote:Expr.2-d g0.14)])
                      (if (and (not (constant? d)))
                          (let-values () (rhs.15 d))
                          (make-L1:quote:Expr.12
                            (Lsrc:quote:Expr.2-d g0.14))))]
                   [(eqv? tag 2)
                    (make-L1:if:Expr.13
                      (Expr (Lsrc:if:Expr.3-e0 g0.14))
                      (Expr (Lsrc:if:Expr.3-e1 g0.14))
                      (Expr (Lsrc:if:Expr.3-e2 g0.14)))]
                   [(eqv? tag 3)
                    (make-L1:begin:Expr.14
                      (map (lambda (m) (Expr m))
                        (Lsrc:begin:Expr.4-e* g0.14))
                      (Expr (Lsrc:begin:Expr.4-e g0.14)))]
                   [(eqv? tag 4)
                    (make-L1:lambda:Expr.15
                      (Lsrc:lambda:Expr.5-x* g0.14)
                      (Expr (Lsrc:lambda:Expr.5-body g0.14)))]
                   [(eqv? tag 5)
                    (make-L1:let:Expr.16
                      (Lsrc:let:Expr.6-x* g0.14)
                      (map (lambda (m) (Expr m))
                        (Lsrc:let:Expr.6-e* g0.14))
                      (Expr (Lsrc:let:Expr.6-body g0.14)))]
                   [(eqv? tag 6)
                    (make-L1:letrec:Expr.17
                      (Lsrc:letrec:Expr.7-x* g0.14)
                      (map (lambda (m) (Expr m))
                        (Lsrc:letrec:Expr.7-e* g0.14))
                      (Expr (Lsrc:letrec:Expr.7-body g0.14)))]
                   [(eqv? tag 7)
                    (make-L1:set!:Expr.18
```

```
                        (Lsrc:set!:Expr.8-x g0.14)
                        (Expr (Lsrc:set!:Expr.8-e g0.14)))]
                      [(eqv? tag 8)
                       (make-L1:pr:Expr.19
                        (Lsrc:pr:Expr.9-pr g0.14)
                        (map (lambda (m) (Expr m))
                          (Lsrc:pr:Expr.9-e* g0.14)))]
                      [(eqv? tag 9)
                       (make-L1:call:Expr.20
                        (Expr (Lsrc:call:Expr.10-e g0.14))
                        (map (lambda (m) (Expr m))
                          (Lsrc:call:Expr.10-e* g0.14)))]
                      [else
                       (error 'convert-complex-datum
                        "unexpected Expr"
                        g0.14)]))])))))))
    (let ([x (let-syntax ([quasiquote '#<procedure>]
                          [in-context '#<procedure>])
               (begin
                 (let ([x (Expr x)])
                   (if (null? const-x*)
                       x
                       `(let ([,const-x* ,const-e*] ...) ,x)))))])
      (unless ((lambda (x) (or (L1:Expr.11? x) (uvar? x))) x)
        (error 'convert-complex-datum
          (format "expected ~s but got ~s" 'Expr x)))
      x)))
```

This exposes the code generated by `define-pass` but does not expand the language form construction templates. The autogenerated clauses, such as the one that handles `set!`, have a form like the following:

```
[(eqv? tag 7)
 (make-L1:set!:Expr.18
   (Lsrc:set!:Expr.8-x g0.14)
   (Expr (Lsrc:set!:Expr.8-e g0.14)))]
```

Here, the tag of the record is checked and a new output-language record constructed, after recurring to the `Expr` transformer on the `e` field.

The body code also changes slightly, so that the output of the pass can be checked to make sure that it is a valid `L1 Expr`.

In addition to echoing the output of the entire pass, it is also possible to echo just the expansion of a single transformer by prefixing the transformer with the `echo` keyword.

```
(define-pass convert-complex-datum : Lsrc (x) -> L1 ()
  (definitions
    (define const-x* '())
```

```
    (define const-e* '())
    (define datum->expr
      (with-output-language (L1 Expr)
        —)))
  (echo Expr : Expr (ir) -> Expr ()
    [(quote ,d)
     (guard (not (constant? d)))
     (let ([t (unique-name 't)])
       (set! const-x* (cons t const-x*))
       (set! const-e* (cons (datum->expr d) const-e*))
       t)])
  (let ([x (Expr x)])
    (if (null? const-x*)
        x
        `(let ([,const-x* ,const-e*] ...) ,x))))
```

⇒

```
Expr in pass convert-complex-datum expanded into:
(define Expr
  (lambda (ir)
    (let ([g141.165 ir])
      (let-syntax ([quasiquote '#<procedure>]
                   [in-context '#<procedure>])
        (begin
          (let ([rhs.166 (lambda (d)
                           (let ([t (unique-name 't)])
                             (set! const-x* (cons t const-x*))
                             (set! const-e*
                               (cons (datum->expr d) const-e*))
                             t))])
            (cond
              [(uvar? g141.165) g141.165]
              [else
               (let ([tag (nanopass-record-tag g141.165)])
                 (cond
                   [(eqv? tag 1)
                    (let* ([d (Lsrc:quote:Expr.2-d g141.165)])
                      (if (and (not (constant? d)))
                          (let-values () (rhs.166 d))
                          (make-L1:quote:Expr.12
                            (Lsrc:quote:Expr.2-d g141.165))))]
                   [(eqv? tag 2)
                    (make-L1:if:Expr.13
                      (Expr (Lsrc:if:Expr.3-e0 g141.165))
                      (Expr (Lsrc:if:Expr.3-e1 g141.165))
                      (Expr (Lsrc:if:Expr.3-e2 g141.165)))]
                   [(eqv? tag 3)
                    (make-L1:begin:Expr.14
                      (map (lambda (m) (Expr m))
                           (Lsrc:begin:Expr.4-e* g141.165))
                      (Expr (Lsrc:begin:Expr.4-e g141.165)))]
                   [(eqv? tag 4)
                    (make-L1:lambda:Expr.15
                      (Lsrc:lambda:Expr.5-x* g141.165)
```

```
   (Expr (Lsrc:lambda:Expr.5-body g141.165)))]
[(eqv? tag 5)
 (make-L1:let:Expr.16
   (Lsrc:let:Expr.6-x* g141.165)
   (map (lambda (m) (Expr m))
        (Lsrc:let:Expr.6-e* g141.165))
   (Expr (Lsrc:let:Expr.6-body g141.165)))]
[(eqv? tag 6)
 (make-L1:letrec:Expr.17
   (Lsrc:letrec:Expr.7-x* g141.165)
   (map (lambda (m) (Expr m))
        (Lsrc:letrec:Expr.7-e* g141.165))
   (Expr (Lsrc:letrec:Expr.7-body g141.165)))]
[(eqv? tag 7)
 (make-L1:set!:Expr.18
   (Lsrc:set!:Expr.8-x g141.165)
   (Expr (Lsrc:set!:Expr.8-e g141.165)))]
[(eqv? tag 8)
 (make-L1:pr:Expr.19
   (Lsrc:pr:Expr.9-pr g141.165)
   (map (lambda (m) (Expr m))
        (Lsrc:pr:Expr.9-e* g141.165)))]
[(eqv? tag 9)
 (make-L1:call:Expr.20
   (Expr (Lsrc:call:Expr.10-e g141.165))
   (map (lambda (m) (Expr m))
        (Lsrc:call:Expr.10-e* g141.165)))]
[else
 (error 'convert-complex-datum
   "unexpected Expr"
   g141.165)]))]))))))))
```

2.3.6.4. Tracing passes and transformers. Echoing the code generated by `define-pass` can help compiler writers to understand what is happening at expansion time, but it does not help in determining what is happening at run time. To facilitate this type of debugging, passes and transformers can be traced at run time. The tracing system, similar to Chez Scheme's `trace-define-syntax`, unparses the input-language term and output-language term of the pass using the language unparsers to provide the S-expression representation of the language term that is being transformed.

The `trace-define-pass` form works just like the `define-pass` form but adds tracing for the input-language term and output-language term of the pass. For instance, if we want to trace the processing of the input:

```
(car (vector-ref '#((1 2) 3) 0))
```

the pass can be defined as a tracing pass, as follows:

```
(trace-define-pass convert-complex-datum : Lsrc (x) -> L1 ()
  (definitions
    (define const-x* '())
    (define const-e* '())
    (define datum->expr
      (with-output-language (L1 Expr)
        —)))
  (Expr : Expr (ir) -> Expr ()
    [(quote ,d)
     (guard (not (constant? d)))
     (let ([t (unique-name 't)])
       (set! const-x* (cons t const-x*))
       (set! const-e* (cons (datum->expr d) const-e*))
       t)])
  (let ([x (Expr x)])
    (if (null? const-x*)
        x
        `(let ([,const-x* ,const-e*] ...) ,x))))
```

Running the class compiler with `convert-complex-datum` traced produces the following:

```
> (np-compile '(car (vector-ref '#((1 2) 3) 0)))
|(convert-complex-datum (car (vector-ref '#((1 2) 3) '0)))
|(let ([t.1 (let ([t.2 (make-vector '2)])
              (begin
                (vector-set! t.2 '0 (cons '1 (cons '2 '())))
                (vector-set! t.2 '1 '3)
                t.2))])
   (car (vector-ref t.1 '0)))
1
```

The tracer prints the *pretty* (i.e., S-expression) form of the language, rather than the record representation, to allow the compiler writer to read the terms more easily. This does not trace the internal transformations that happen within the transformers of the pass. Transformers can be traced by adding the `trace` keyword in front of the transformer definition. We can run the same test with a `convert-complex-datum` that traces the `Expr` transformer, as follows:

```
(define-pass convert-complex-datum : Lsrc (x) -> L1 ()
  (definitions
    (define const-x* '())
    (define const-e* '())
    (define datum->expr
      (with-output-language (L1 Expr)
        —)))
  (trace Expr : Expr (ir) -> Expr ()
    [(quote ,d)
     (guard (not (constant? d)))
     (let ([t (unique-name 't)])
       (set! const-x* (cons t const-x*))
       (set! const-e* (cons (datum->expr d) const-e*))
```

```
        t)])
  (let ([x (Expr x)])
    (if (null? const-x*)
        x
        `(let ([,const-x* ,const-e*] ...) ,x))))
```

```
> (np-compile '(car (vector-ref '#((1 2) 3) 0)))
|(Expr (car (vector-ref '#((1 2) 3) '0)))
| (Expr (vector-ref '#((1 2) 3) '0))
| |(Expr '#((1 2) 3))
| |t.1
| |(Expr '0)
| |'0
| (vector-ref t.1 '0)
|(car (vector-ref t.1 '0))
1
```

Because the `datum->expr` procedure handles the heavy lifting of producing the expression for creating the constant, the trace for the quoted vector simply takes the quoted vector and returns the new temporary generated for it. Here, too, the traced forms are the pretty representation and not the record representation.

2.4. Comparison with the Prototype Nanopass Framework

2.4.1. Syntactic differences. The most obvious differences between the prototype framework developed by Sarkar et al. and the one described in this dissertation are the syntactic differences. The changes to language definitions are largely aesthetic and simplify the parsing performed by the macro that defines the `define-language` form. The `terminals` section is now explicitly defined within the S-expression `terminals` form, rather than implicitly between the `over` and `where` keywords. The `in` keyword also is dispensed with; instead, the terminal type or nonterminal name is followed by a parenthesized list of meta-variables to represent the language. In an extended language definition, such as that seen in Figure 2.2, the `extends` keyword is now parenthesized, and the `+` and `-` keywords are moved within the terminals or nonterminal sections and parenthesized.

Pass definitions also have a slightly different syntax. These changes support the addition of extra formals and extra return values to a pass. The input-language-term formal to a pass or a transformer is now bound explicitly. As seen in Figure 2.3, the new format uses the variable `ir` to represent the input expression. Explicitly binding the input expression to a visible variable also allows the input-language term to be referenced in the `else` clause or in the body of the pass. The new pass syntax also allows for the input and output nonterminal entry point to be specified. This is accomplished by listing the language and nonterminal names together in place of the language name.

```
(define-language L0 over
  (x in variable)
  (b in boolean)
  (n in integer)
 where
  (Program
    Expr)
  (e body in Expr
    b
    n
    x
    (if e1 e2 e3)
    (seq c1 e2) => (begin c1 e2)
    (lambda (x ...) body)
    (e0 e1 ...))
  (c in Command
    (set! x e)
    (seq c1 c2) => (begin c1 c2)))
```

```
(define-language L0
  (terminals
    (variable (x))
    (boolean (b))
    (integer (n)))
  (Program (prog)
    e)
  (Expr (e body)
    b
    n
    x
    (if e1 e2 e3)
    (seq c1 e2) => (begin c1 e2)
    (lambda (x ...) body)
    (e0 e1 ...))
  (Command (c)
    (set! x e)
    (seq c1 c2) => (begin c1 c2)))
```

FIGURE 2.1. Comparing a language definition between the prototype and new nanopass frameworks.

```
(define-language L1 extends L0
  over
   - (b in boolean)
  where
   - (Expr b (if e1 e2 e3))
   + (default in Expr
       (case x (n1 e1) ...
         default)))
```

```
(define-language L1
  (extends L0)
  (terminals
    (- (boolean (b))))
  (Expr (e body default)
    (- b
       (if e1 e2 e3))
    (+ (case x (n1 e1) ...
         default))))
```

FIGURE 2.2. Comparing language extension between the prototype and new nanopass frameworks.

For instance, in place of `L0` the term `(L0 Command)` indicates that the pass should expect a `Command` nonterminal. The catamorphism syntax is also extended from the prototype nanopass framework to support additional arguments to transformers.

2.4.2. Semantic differences. The prototype nanopass framework defines the meta-parser procedure for a language as a compile-time (or meta) definition. The procedure is then retrieved using `eval` when a pass definition is encountered. This means that languages can be defined only at the top level of a program or library. The new nanopass framework allows language definitions in any scope by making the meta-parser available through the compile-time environment. This allows the procedure to be found wherever the identifier for the language is in scope.

```
(define-pass remove-implicit-begin L0 -> L1
  (process-expr-expr : Expr () -> Expr ()
    [(lambda (,x ...) ,[body1] ... ,[body2])
     `(lambda (,x ...) (begin ,body1 ... ,body2))]
    [(let ((,x ,[e]) ...) ,[body1] ... ,[body2])
     `(let ((,x ,e) ...) (begin ,body1 ... ,body2))]
    [(letrec ((,x ,[e]) ...) ,[body1] ... ,[body2])
     `(letrec ((,x ,e) ...) (begin ,body1 ... ,body2))]))

(define-pass remove-implicit-begin : L0 (ir) -> L1 ()
  (process-expr-expr : Expr (ir) -> Expr ()
    [(lambda (,x ...) ,[body1] ... ,[body2])
     `(lambda (,x ...) (begin ,body1 ... ,body2))]
    [(let ((,x ,[e]) ...) ,[body1] ... ,[body2])
     `(let ((,x ,e) ...) (begin ,body1 ... ,body2))]
    [(letrec ((,x ,[e]) ...) ,[body1] ... ,[body2])
     `(letrec ((,x ,e) ...) (begin ,body1 ... ,body2))]))
```

FIGURE 2.3. Comparing pass definitions between the prototype and new nanopass frameworks.

In the prototype nanopass framework, each pass takes a single input-language term and returns a single output-language term value. This approach requires that information gathered during an analysis pass be encoded in the intermediate representation when it is needed in another pass (or stored in a mutable variable that is visible to both passes). In a commercial compiler, however, it can be useful to keep information separate from the intermediate representation. To support this, the new nanopass framework allows passes to take extra arguments and return extra values.

Passes defined in the prototype nanopass framework operate on a full language term, but it can be useful to operate on a language term that starts from a different nonterminal. For instance, we might write a set of passes that operate over the functions defined in a program term, without wanting to pattern match the full program term in each pass.

The prototype and new framework also handle the extra return values for transformers differently. In the prototype framework, each extra return value requires a procedure to combine the results from recurring on sub-nonterminals. In a compiler like the one developed in the compiler course, this is useful for combining lists, such as the lists of free variables gathered during free-variable analysis, or for merging live variable information during live variable analysis. In a commercial compiler, we want to avoid using this kind of representation; instead, values are threaded through the transformer and updated non-destructively along the way as new information is learned. Instead of using an environment, we modify information directly in a mutable field in the representation of a variable. The new nanopass framework allows any expression to appear as the extra return value.

The prototype nanopass framework required every transformer, even those that do not require user-specified clauses, to be named by the compiler writer. The new framework can autogenerate a processor when there is a single input-language term and a single output-language term. Autogeneration is driven by need. When no transformer is found in the lookup for a catamorphism or autogenerated clause, a new transformer is generated.

In the new nanopass framework, passes can omit the input-language term or output-language term. A pass can also start processing over a non-entry nonterminal by specifying that the nonterminal entry point for the pass is part of the signature of the pass.

Two new syntactic forms, `nanopass-case` and `with-output-language`, are also provided by the new nanopass framework. The `nanopass-case` form allows a language form to be matched outside of the left-hand side of a pass clause. This can be useful to generate predicates over language terms or to perform further matching on a sub-form. The `nanopass-case` form is syntactic sugar for the `define-pass` form.

The prototype nanopass framework allows definitions only at the pass level, but the new nanopass framework also provides a `definitions` clause within each transformer. When used within a transformer, the `quasiquote` transformer is bound to allow language terms in the output nonterminal to be constructed.

The prototype nanopass framework restricted the syntax of the `define-pass` body to contain a single `let-values` expression or an empty list. In the new nanopass framework, the body of a pass can contain zero or more expressions. When the body contains no expressions, the `define-pass` form autogenerates the body based on the input and output nonterminals specified by the pass, either explicitly, by using the language and nonterminal name as the input or output-language specifier, or implicitly, by specifying the language name and the entry nonterminals for the input and output language being used. The autogenerated body calls or autogenerates the appropriate transformer for the expected input-language term.

When `unquote` is used in the templates on the right-hand side of a transformer clause, the context of the `quasiquote` is not updated to expect the field type at that location. Instead, the compiler writer must explicitly change the context using the `in-context` transformer. The new nanopass framework automatically puts the expression in an `unquote` form into the correct context.

The new nanopass framework also adds an `else` clause to the possible clauses of a transformer. This allows the compiler writer to prevent the autogeneration of clauses in this transformer. In a

commercial compiler, this is an important feature when a transformation should be performed only on a given form and no automatic recursion is needed, or when writing a predicate that matches only a few productions.

The `with-output-language` form allows a language form to be constructed outside of the right-hand side of a pass clause. This can be useful for functional abstraction and for constructing terms from an incoming source, such as a parser or the output of a syntax expander. Unlike the `nanopass-case` form, the `with-output-language` form is not syntactic sugar for `define-pass`. Instead, it simply creates the same binding for `quasiquote` as in the body of a transformer. This allows `with-output-language` to act as a splicing form, so multiple definitions can be put within the form and are still scoped at the same level as the `with-output-language` form.

2.4.3. Engineering improvements. In addition to semantic and syntactic changes, the new nanopass framework performs more error checking and produces more meaningful error messages. For instance, meta-variables are checked for uniqueness. When an error can be determined during expansion, an exception is immediately raised instead of waiting until run time. When an exception might occur at run time, every attempt is made to provide the file position information for the location from where the error originates.

The prototype nanopass framework maintains two representations of the language in the compile-time environment. The first is an S-expression representation of the information parsed from the `define-language` form, and the second is an annotated version of the first that includes the language-form record information. The `define-pass` macro looks up this information in the compile-time environment and converts it to a record representation that it uses for the duration of the macro. The new nanopass framework avoids the double representation and the conversion from the annotated S-expressions into a record representation by simply representing the language information as a set of records. Part of the reason that the prototype nanopass framework uses an S-expression representation is to allow the expander to rewrite the marks and substitutions on identifiers, such as the terminal predicates, which are not rewritten when stored in records. The new nanopass framework instead attaches an extra mark to these identifiers, via a local macro, to preserve the marks.

The prototype nanopass framework uses two different methods to determine the target of a catamorphism or the recurrence of a sub-nonterminal of an autogenerated clause. For a catamorphism, the target is determined by finding a transformer with an input type that matches the input field type, and an output type that matches the specified output variable type that also supplies the correct

number of extra return values. For an autogenerated clause, the target is determined solely by the input field type. The new nanopass framework uses a common procedure to determine the target of the recursion. It also uses any extra arguments supplied (in the case of a catamorphism) or extra formals available in the calling transformer (in the case of an autogenerated recursion or catamorphism where the arguments are not explicitly supplied) and any extra return values expected to help narrow down the target. In the new nanopass framework, the lookup procedure is also responsible for automatically generating new transformers when an appropriate transformer is not supplied by the compiler writer. This allows transformer autogeneration to be driven by need.

The new nanopass framework also uses an integer tag in place of the record dispatch used in the prototype framework. This can slightly speed record matching by replacing a potentially looping record predicate with an efficient `fx=?` predicate.

2.5. Related Work

Stratego/XT [**16**] is a DSL for writing source-to-source transformations. It provides a set of combinators for matching and rebuilding abstract syntax tree (AST) forms as well as strategies for performing different traversals of the AST. Stratego does not differentiate between pattern matching and AST construction combinators, and this leads to a matching semantics that is different from that of the nanopass framework. In particular, if an error occurs when constructing an output term, the pattern match is treated as a failure, and the next pattern is tried. This can make debugging more difficult, as the final pattern in a set of matches is often the one that raises an error, even though the error might have been caused by the construction part of another clause. The internal languages are also not checked to verify that the grammar is followed, as they are in the nanopass framework. The combinators and strategies are not without their benefits, however, and transformations can be written in small pieces and composed into larger transformations.

The JastAdd [**43**] system allows for the construction of modular and extensible compilers using Java's object-oriented class hierarchy, along with an external DSL to specify the abstract syntax tree and analysis and transformations on these trees. In JastAdd, each type of node in the AST has an associated class that encapsulates the transformations for that node type. Classes implement an attribute grammar. Instead of AST pattern matching, as provided by the nanopass framework, method dispatch and aspects are used for implementing a given compiler class with the visitor pattern. We believe that there is an advantage to providing pattern matching, in that it simplifies

the expression of transformations that need to look at the children of a given node, and provides a compact syntax for what would otherwise require a local tree traversal.

SableCC [**49**] is a system for building compilers and interpreters in Java. It provides a full set of tools for writing the lexer and parser for the language and a method for defining multiple passes. Similar to JastAdd, it works on an AST represented by Java classes. Also similar to JastAdd, it uses the visitor pattern to implement the language transformations.

POET combines a transformation language and an empirical testing system to allow transformation to be tuned [**97**]. Although POET does allow for some generic manipulation of an AST, it is largely focused on targeting specific regions of source code to be tuned. To this end, it provides a language for specifying the parsing of fragments of source code and then acts on these AST fragments, preserving unparsed code across the transformation. The nanopass framework does not deal with concrete syntax; instead, it relies on the internal representations and parsing to and from S-expressions.[3]

The ROSE [**71**] compiler infrastructure provides a C++ library for source-to-source transformation, along with front-ends and back-ends for both C/C++ and Fortran code. Internally, ROSE represents source code using the ROSE Object-Oriented IR, with transformations written in C++. Although there is nothing specific that ties the ROSE transformation framework to C/C++ or Fortran, there are no tools to easily add new front-ends and back-ends, limiting the usefulness of ROSE for other languages.

The Rhodium framework takes a different approach from some of the other tools described here. Instead of building from term rewriting, Rhodium bases its transformations on data-flow equations and provides a framework for proving the soundness of transformations [**67**, **75**]. The framework has also been extended to support inferring optimizations from the data-flow semantics defined by the compiler writer. The data-flow facts are defined over a C-like intermediate representation. A tool such as this might be a good complement to the nanopass framework, but it would require the compiler writer to specify both the static semantics of the input and output language and the data-flow facts.

CodeBoost [**14**] is an example of a more targeted tool. Although the tool was originally developed to support the Sophus numerical library, CodeBoost provides a fairly simple way to do compiler transformations within C++ code. It is implemented using Stratego but provides an array of tools to make writing C++ code transformations easier. The focus on C++ makes this a much more focused tool than the nanopass framework is intended to be.

[3]Of course, in a Scheme compiler, S-expressions are the concrete syntax.

The template-based metacompiler (Tm) [**72**] provides a macro language, similar to M4, for generating data structures and transformations that can be expanded in a language agnostic way. Tm provides tree-walker and analyzer templates and an existing C back-end for easier use.

Pavilion [**96**] is a DSL for writing analysis and optimization passes to improve the efficiency of generic programming in C++. The declarative language extends regular expressions with intersection and complement operators, variable quantification, path quantification, function definition, and native language access to Scheme to provide powerful matching during analysis and transformation.

Yoko [**47**] is a Haskell module for writing functions that transform an input type to a similar output type that requires that only the interesting cases be specified, similar to how the nanopass framework operates over languages. The module builds on generic programming techniques to provide the `hcompos` function. The `hcompos` function takes the user-specified cases and autogenerates the necessary clauses to handle constructors from the input type not specified by the programmer, matching them with constructors of the output type with the same name. Because these transformations are based on Haskell types, functions created in this way are guaranteed to generate terms with the appropriate output type by the type checker.

CHAPTER 3

Evaluating the Nanopass Framework

3.1. Introduction

To create a nanopass framework capable of developing a commercial compiler, we need to make a commercial compiler both to serve as a proof of concept for the framework and to provide feedback to the development of the framework. Thus, we decided to develop and test the nanopass framework by replacing the original compiler for Chez Scheme [**33**] with a new compiler. The new compiler needs to support all the features, e.g., a complete Revised^6 Report on Scheme (R6RS) implementation, a foreign function interface, a thread system, and debugging and profiling tools, of the existing Chez Scheme compiler. It also needs to generate code that is on par with the original compiler. The new compiler makes use of the existing run-time system and Scheme libraries to give us a good starting point for the new compiler.

Although we could have created a compiler that generates exactly the same machine code as the existing compiler, developing a new compiler seemed an excellent opportunity to improve on existing optimizations and to try some different techniques, such as replacing the register allocator with a graph-coloring register allocator. Although some of the features the new compiler implements are not as fast as those of the original compiler, we still wanted to keep the speed of the compiler within a factor of two of the original compiler.

Developing a new commercial compiler also directed how the nanopass framework needed to be improved to support this effort. As we went through the development process, features missing from the prototype framework, such as the `with-output-language` form, were added to the new nanopass framework. This also led to feature changes, such as better support for additional arguments in the catamorphism syntax. Additionally, some features, such as the automatic combining of extra return values from the prototype nanopass framework, were removed.

3.2. Transitioning a Class Compiler

The class compiler was originally written using an S-expression representation and a match form that supports catamorphisms to implement the passes. To see how a nanopass framework version of the class compiler compares, the compiler was re-implemented over the course of a weekend.

The compiler is comprised of 38 passes. It compiles a small subset of Scheme to x86_64 assembly code. The compiler uses a graph-coloring register allocator and contains a few optimizations, such as optimizing known calls and optimizing away jumps to jumps. The assembly code is then either interpreted, through a simple assembly language interpreter written to safely test student compilers, or compiled with a simple C run time into a stand-alone program.

The class compiler also has a testing framework that evaluates the output of each pass and compares the result with the value of the original expression evaluated in the host compiler, in this case, Chez Scheme. The nanopass version of the class compiler, as it was a weekend project, does not support the testing between passes. As a result, the testing is performed with the evaluation of pass output disabled in the original compiler so that the two are performing the same tasks.

The code sizes of the two compilers are compared by using the Scheme reader to read the source files and the pretty printer to print them to a consistent line length. The number of lines of each expression is then counted to compare the relative lengths. The code for the nanopass compiler (excluding the common helpers, drivers, and assembly language interpreter used by both) is 18% shorter than the code for the original class compiler. Comparing the passes directly, the nanopass compiler is 35% shorter. When the intermediate language definitions are taken into account, the nanopass compiler with passes and languages is 21% shorter.

The relative speed of the two compilers is also compared on a small set of tests used during the class to ensure that the student compilers run correctly. The tests were performed using the assembly language interpreter for both compilers to avoid the round trip to the file system and the cost of running the external GCC compiler, which could skew the results. The nanopass version compiles and interprets the assembly code of the tests 52% faster.

3.3. Recreating a Commercial Compiler

3.3.1. Background on Chez Scheme. Chez Scheme is a commercial Scheme compiler for R6RS Scheme with extensions, first released in 1985 [**33**] and under continuous development and improvement over the last 27 years [**5**, **12**, **18**–**21**, **30**–**32**, **35**–**41**, **53**–**56**, **59**, **60**, **91**, **93**]. The compiler is

written in Scheme and is comprised of a small number of large, multipurpose passes. The compiler is almost *absurdly* fast, able to compile its own source code in roughly three seconds on contemporary hardware. The compiler is also designed to generate efficient code.

3.3.2. Workings of the existing Chez Scheme compiler. The Chez Scheme compiler begins with the `syntax-case` expander, extended with a module system and R6RS libraries [**34**, **40**, **55**, **91**]. The result of the expander is a simplified core language with `letrec`, `letrec*`, and `case-lambda` as binding forms, quoted constants, primitive references, procedure calls, variable references, and a handful of other forms. This simplified form records source information used for debugging and profiling, through the pass described in Section 3.4.1. The next pass places validity checks for variable references bound by `letrec` and `letrec*` [**56**, **93**]. The source optimizer [**90**] discussed in Section 3.4.2 is the next pass. The source optimizer can be run one or more times or not at all, depending on the options set in the compiler. A pass for handling `letrec` and `letrec*` [**56**, **93**] is run at least once and is run between each run of the source optimizer. After this, either the interpreter is invoked to interpret the program or the back-end compiler is invoked to finish compilation and either execute the resulting code or write the results to the file system.

When the back-end compiler is invoked, the code is still in roughly the same form as the input language, although `letrec*` has been eliminated and all `letrec` bindings now bind only unassigned variables to `case-lambda` expressions. The compiler next performs assignment conversion, closure conversion, various optimizations, and further code simplifications and replaces primitives in the source language with an internal set of simpler, although still higher level than assembly language, primitives. It also performs register allocation using a linear-scan style register allocator with a lazy register save and restore strategy [**21**], and it generates code using destination-driven code generation [**41**].

The back-end consists of five passes. The first pass, `cp1`, begins the process of closure conversion, recognizes loops, recognizes direct application of λ-expressions, begins handling multiple return value calls, sets up the foreign and foreign callable expressions, and converts primitive calls into a set of slightly lower-level inline calls. The next pass, `cpr0`, begins the register allocation process, determines the actual free variables of closures after performing closure optimization, performs assignment conversion, makes explicit all arguments to inlined primitives, and flags tail calls and loops. The register allocator reserves the architectural stack register, a Scheme frame pointer register, a thread context register pointer, and at least two temporary registers, depending on the target machine architecture, for use by the assembler. After this, the pass `cpr1` assigns variables to their

initial register homes. The `cpr2` pass finishes register allocation, generating register saves and restores across non-tail calls and removing redundant register bindings. Finally, the `cp2` pass finishes compilation, converting the higher-level inlined primitives into a set of high-level instruction inlines that the assembler can convert into machine instructions for the target platform.

3.3.3. Workings of the new compiler. The new compiler starts with the same set of front-end passes described in the first paragraph of the preceding section. These passes implement the same algorithms as the ones in the original compiler but are updated to use the nanopass framework. A comparison of two of these passes is provided in Section 3.4.

The back-end of the new compiler diverges significantly from that of the original compiler. Where the original compiler is structured as a set of multipurpose passes that each performs several tasks, the new compiler is organized as many smaller passes, with each pass completing primarily one task. Overall there are approximately 50 passes and approximately 35 nanopass languages.

These passes implement most of the optimizations from the original compiler and improve on some. These improvements include support for implicit cross-library optimization, which is described along with the `library-group` form in Chapter 4. The closure optimization process has also been improved substantially and is described in detail in Chapter 5.

Outside of these optimizations, there is also improved handling of procedures that return multiple-values. Both the original compiler and the new compiler attempt to push the consumer into the producer, when the producer is a λ-expression [**12**]. In both, when a `values` expression is found in the tail position of the producer code, an `mvlet` is used, similar to the way that the direct application of a λ-expression is recognized as a `let`. This allows both compilers to avoid creating a closure for the consumer in these cases.

The original compiler cannot push the consumer into the tails of an `if` expression because it would require duplication of the consumer code. The new compiler does push the consumer into the tails of an `if` expression by creating a label for the consumer code and using a `goto` to jump directly to the consumer code without needing to create a closure.

The new compiler can also efficiently handle chained multiple return value calls, where a consumer is also a multiple value producer for a following consumer. This works by having each producer position its return values in the same order as the arguments for the next call, even though they might need to be shifted on the frame if the final call in the chain is a tail call.

The other big difference between the original compiler and the new compiler is the use of a graph-coloring register allocator. This change necessitates several other changes in the compiler, including the expansion of code into a near-assembly language form much earlier in the compiler. This expansion is required so that all of the temporaries that might be needed for the final code to conform to the operand requirements of the machine can be met. It also means that a full live analysis must be performed to compute the conflict graph needed by the register allocator. In the original compiler, the cost of the live analysis is largely avoided by tracking the liveness of registers, rather than the liveness of variables. Additionally, in the original compiler, primitive expansion is delayed until code generation at the cost of reserving two or more registers and, depending on the target architecture, makes these registers unavailable to the register allocator.

The upside of using a graph-coloring register allocator is that it packs spilled variables tighter on the frame, makes better use of registers, and generally produces more compact code. The new register allocator also makes use of move biasing to avoid frame-to-frame moves. This contributes to the generated code's faster run time, at the cost of substantial extra compile-time overhead. Both the original and new compilers also try to make good use of variable saves and restores around non-tail calls. This means call-live variables can be accessed from a register, rather than from a frame variable. The original compiler follows a lazy-save strategy [**21**], while the new compiler attempts to get similar results through using a heuristic that estimates the cost of saving and restoring versus the cost of spilling to a frame location permanently. This is one place where the new compiler underperforms the original compiler.

Not all of the features and optimizations provided by the original compiler are provided by the new compiler. The most significant missing feature of the original compiler is support for 32-bit Sparc, 64-bit Sparc, and PowerPC processors, as the new compiler supports only i386 and x86_64. The most significant missing optimization is block allocation of closures. When several closures are created at the same time, a single allocation is performed to allocate the space for the entire group of closures. A more general block allocation optimization is planned for the new compiler but has not yet been implemented.

3.3.4. Ensuring compatibility. Over the course of Chez Scheme's development, an extensive suite of tests, including regression tests, has been developed to ensure that the compiler conforms to the relevant Scheme standards and that fixed bugs do not recur. Chez Scheme is also bootstrapped, and the first test of the compiler is to compile itself and verify that code generated for the compiler is consistent with each run of the compiler. The new compiler passes all of these tests.

3.4. Comparing Nanopass Passes with Traditional Passes

While a compiler writer might choose the nanopass framework for the convenience of defining languages and passes, using the convenient S-expression syntax, and minimizing boilerplate code, it should not happen at the sacrifice of performance. To see how this compares with a more traditional compiler construction, in this section we compare two passes from Chez Scheme that are similar in the original compiler and new compiler. The first pass records source code and position information in the internal representation of language terms, with a simple, linear pass over the language term. The second pass is the Chez Scheme source optimizer [**90**] that performs aggressive inlining, constant propagation, constant folding, copy propagation, and record optimizations. The source optimizer pass is also linear by design, as the effort and size of each inlining attempt is limited.

The existing Chez Scheme compiler uses an internal representation with tagged vectors and a macro for matching the vectors and binding the elements of the vector to variables. The compiler also uses a simple matcher that allows each variant to be matched, or several variants to be matched at once, if the contents of the vector are not needed.

To test how converting this pass to the nanopass framework affects the run time of the pass, we instrumented the compiler to record the run time of each pass during compilation. We then compiled the benchmarks with the instrumented compiler to record the difference in time between the original compiler and the new compiler. Three benchmarks suites are used for testing: the R6RS benchmarks [**26**]; a set of benchmarks used when testing the source optimizer, which includes some larger programs, and a smaller set of benchmarks used for testing Chez Scheme. There is some overlap between the benchmark suites, as all three include benchmarks such as Tak, Fib, and the Gabriel benchmarks [**48**]. Because the source pass is linear, and many of the benchmarks are short, the run time of this pass can be quite small, so we use a fine-grain timer in the instrumentation to determine how they compare. The measurements are taken on a two CPU, Intel® Core™ i7-3960X with six cores per CPU, running Fedora 17 Linux with the non-threaded version of Chez Scheme. The fine-grain timer uses the `clock_gettime` Linux C library routine to determine the number of nanoseconds used by Scheme, including both user and system time.

3.4.1. Comparing a simple pass. The first step in any R6RS Scheme compiler is macro expansion. In Chez Scheme, the macro expander leaves behind source position information as it runs. Recording of source information is a good start; however, the source information at this point in the compiler does not provide all of the information desired for profiling and debugging purposes.

The source pass records the expanded version of the code in a field on each node of the intermediate representation and replaces the existing variable representation with a new copy of each variable, recording the variable generated during expansion as the source field of the fresh variable.

In the original compiler, every valid form in the internal representation must be visited to update the source fields and to recur through fields of the production that contain other expression nodes. The nanopass framework allows us to rewrite this pass in a shorter form. We can avoid explicitly matching forms that do not contain a source or variable record, as the framework will fill in clauses for the other language forms. When this pass was first written, this provided some reduction in the size of the code, but not as much as we had hoped. Examining the source code for the new version of the source pass, it was evident that the pass could be rewritten in a simpler form, if the nanopass framework allowed transformers to operate over terminals. This is an example of how implementing the new compiler informed the design of the new nanopass framework. After adding the terminal transformer functionality to the nanopass framework, it was possible to add a transformer for handling the source records. This allows us to eliminate any clause that does not contain a variable record. Finally, we can eliminate variable assignment and references by adding another terminal transformer for variable references. In the end the pass only explicitly names the binding forms in the language at this point: `case-lambda`, `letrec`, and `letrec*`.

Table 3.1 presents a comparison of the code size of the two versions of this pass.

TABLE 3.1. Comparing line and character counts for original and nanopass versions of `cpsrc`.

Version	**lines**	**characters**
Original	87	440
Nanopass	55	286

Table 3.2 shows the average of the normalized times on both the 32-bit and 64-bit versions of Chez Scheme at optimize level 2 and optimize level 3. The table also includes the normalized total time for compiling all of the benchmarks. Each benchmark is compiled five times, with a maximum generation collection run between each compile. The times and statistical information are then averaged for that benchmark.

The 32-bit version of the new compiler performs similarly to the original compiler in the time spent in the pass. On average, the new compiler runs the pass at 1.01 times as long at optimize level 2 and optimize level 3. The overall time for compiling all the benchmarks on the 32-bit machine is

TABLE 3.2. Run time of the nanopass version of `cpsrc` relative to the original version.

Optimization level	Machine type	Average relative time	Total relative time
2	x86	1.01	1.07
2	x86_64	1.06	1.10
3	x86	1.01	1.06
3	x86_64	1.05	1.10

slightly slower. The new compiler takes 1.07 times as long at optimize level 2 and takes 1.06 times as long at optimize level 3.

The 64-bit version of the new compiler is slightly slower than the original compiler. On average, the new compiler runs the pass at 1.06 times as long at optimize level 2 and 1.05 times as long at optimize level 3. The overall time for compiling all the benchmarks on the 64-bit machine is slower. The new compiler takes 1.10 times as long at optimize level 2 and optimize level 3. Figure 3.1 presents the normalized results of running the R6RS benchmarks; Figure 3.2 presents the normalized results of running the benchmarks used for testing the source optimizer; and Figure 3.3 presents the normalized results of running the Chez Scheme benchmarks.

When the benchmarks were first tested, the `ddd` benchmark was an outlier in the data, requiring more time to compile on the 64-bit machine by a factor of 7.97 at optimize level 2 and 7.29 at optimize level 3, and on the 32-bit machine at 2.48 at optimize level 2 and optimize level 3. The cause of the high run time of the `cpsrc` pass when compiling `ddd` turned out to be related to garbage collection. Something about the benchmark was tickling the garbage collector in the new compiler in a way that it did not in the original compiler. The effect of the garbage collector was not limited to making the performance worse, as on the 32-bit machine when compiling the R6RS `compiler` benchmark the new compiler runs the `cpsrc` pass at a factor of 0.01 due to garbage collection in the original compiler.

To avoid the effects of the garbage collector, we now perform a maximum collection just before the `cpsrc` pass runs. This helps to stabilize the results, which now range, on the 32-bit version, between a factor of 0.898 and 1.23 at optimize level 2 and between a factor of 0.920 and 1.25 at optimize level 3 and, on the 64-bit version, between a factor of 0.671 and 1.42 at optimize level 2 and between a factor of 0.534 and 1.50 at optimize level 3.

3.4.2. Comparing the source optimizer. The source optimizer pass implements constant propagation, constant folding, copy propagation, and bounded aggressive procedure inlining [**90**]. It also implements record optimization [**63**]. The pass is composed of a set of inliners for primitive folding,

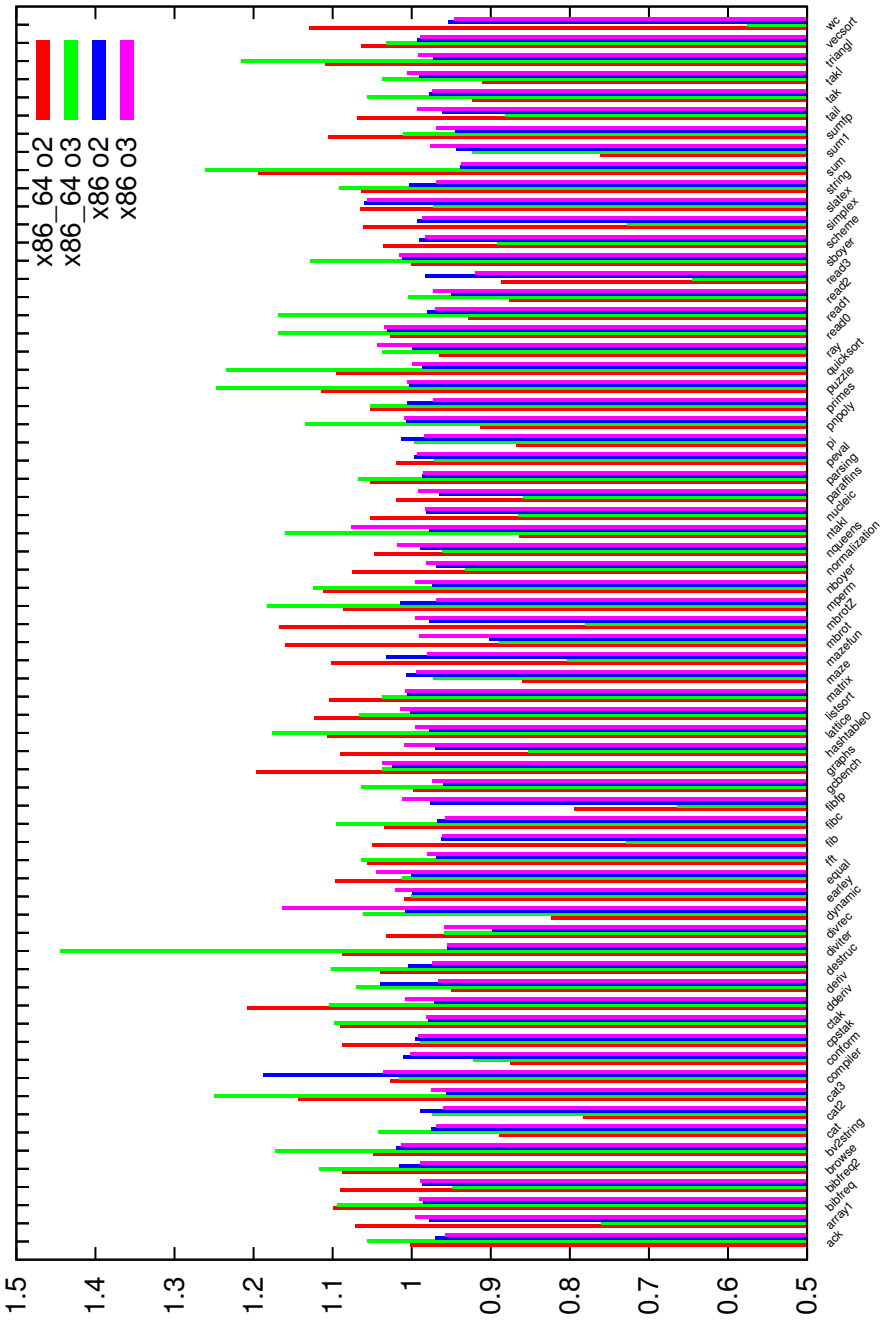

FIGURE 3.1. Normalized run time of `cpsrc` pass for R6RS benchmarks.

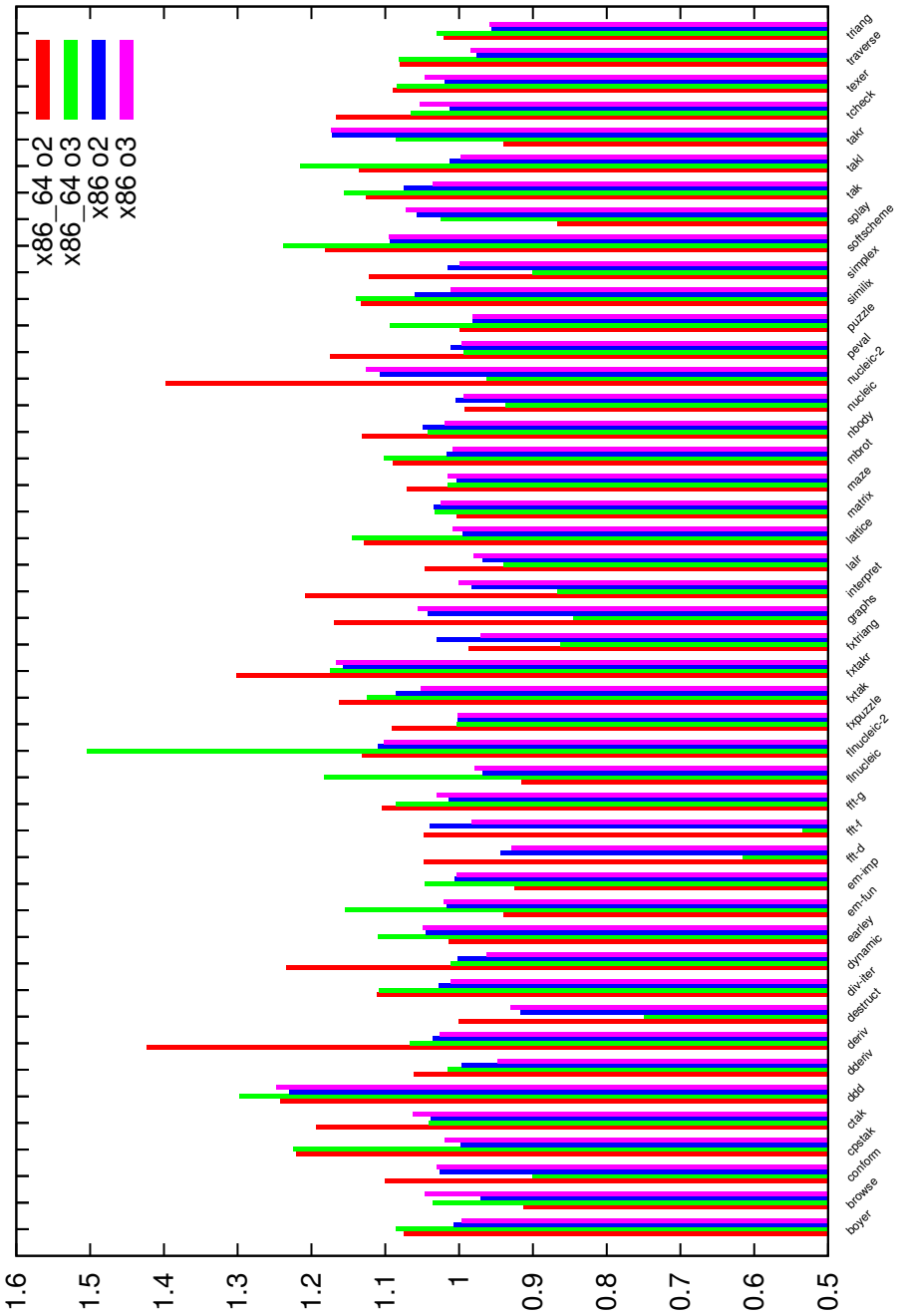

FIGURE 3.2. Normalized run time of `cpsrc` pass for new benchmarks.

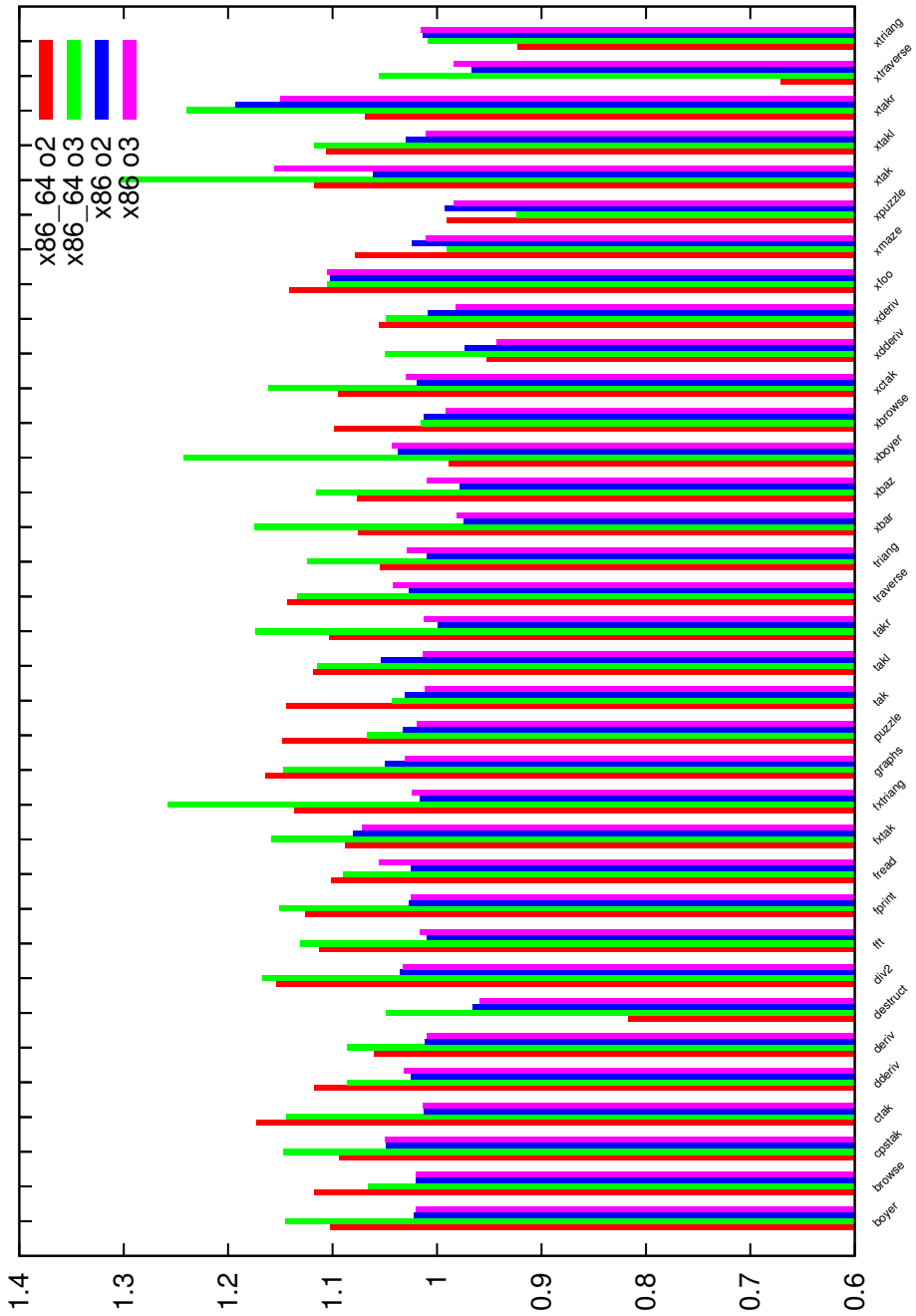

FIGURE 3.3. Normalized run time of `cpsrc` pass for benchmarks.

code to support code copying and procedure inlining, and a set of predicates to determine when expressions are, for example, simple or pure. This means that much of the code is not involved directly in matching language forms; instead, there are smaller amounts of matching surrounded by the code that implements the source inliner. As such, we do not expect much benefit from switching to the nanopass framework, at least in terms of code size, as the matches that need to be performed are essentially the same in both versions of the pass.

Table 3.3 confirms that there is little difference in the size of the two versions of the optimization pass.

TABLE 3.3. Comparing line and character counts for the original and nanopass versions of `cp0`.

Version	**lines**	**characters**
Original	3687	15906
Nanopass	3571	16002

Table 3.4 shows the average of the normalized times on both the 32-bit and 64-bit versions of Chez Scheme at optimize level 2 and optimize level 3. The table also includes the normalized total time for compiling all of the benchmarks. The compiler is run five times on each benchmark, with a maximum generation collection run between each compile. The times and statistical information are then averaged for that benchmark.

TABLE 3.4. Run time of the nanopass version of `cp0` relative to the original version.

Optimization level	**Machine type**	**Average relative time**	**Total relative time**
2	x86	1.17	1.03
2	x86_64	1.22	1.15
3	x86	1.15	0.97
3	x86_64	1.21	1.07

Figure 3.4 presents the normalized results of running the R6RS benchmarks; Figure 3.5 presents the normalized results of running the benchmarks used for testing the source optimizer; and Figure 3.6 presents the normalized results of running the Chez Scheme benchmarks.

On the 32-bit version of Chez Scheme, the average normalized time is 1.17 at optimize level 2 and 1.15 at optimize level 3. This indicates that there is a bit of expense in the nanopass version of the pass. If we total the numbers, however, and normalize them, we see that the total time for running this pass is closer to on par with the original compiler with the nanopass version running at a factor of 1.03 at optimize level 2 and running at a factor of 0.97 at optimize level 3.

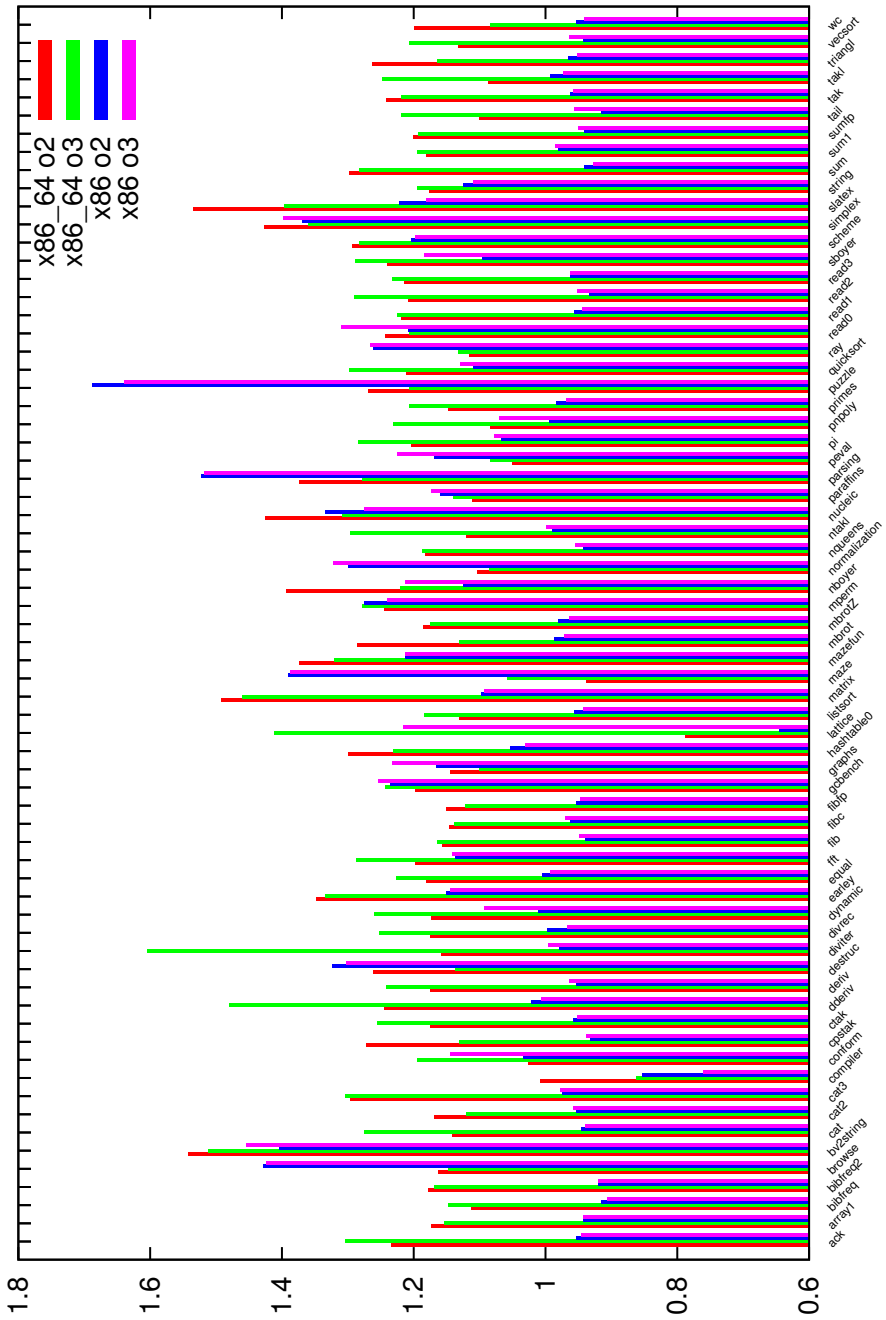

FIGURE 3.4. Normalized run time of `cp0` pass for the R6RS benchmarks.

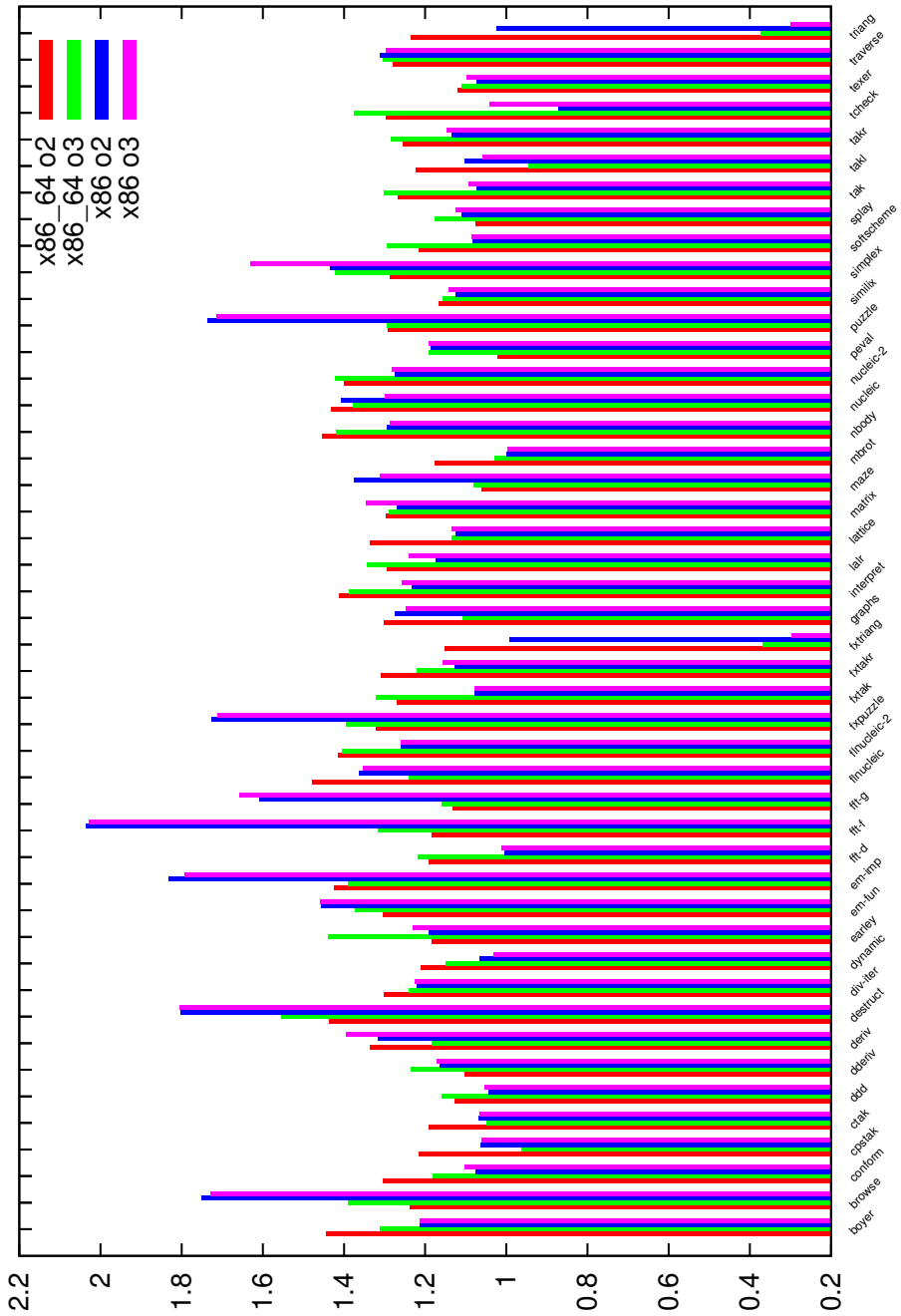

FIGURE 3.5. Normalized run time of `cp0` pass for the source optimizer benchmarks.

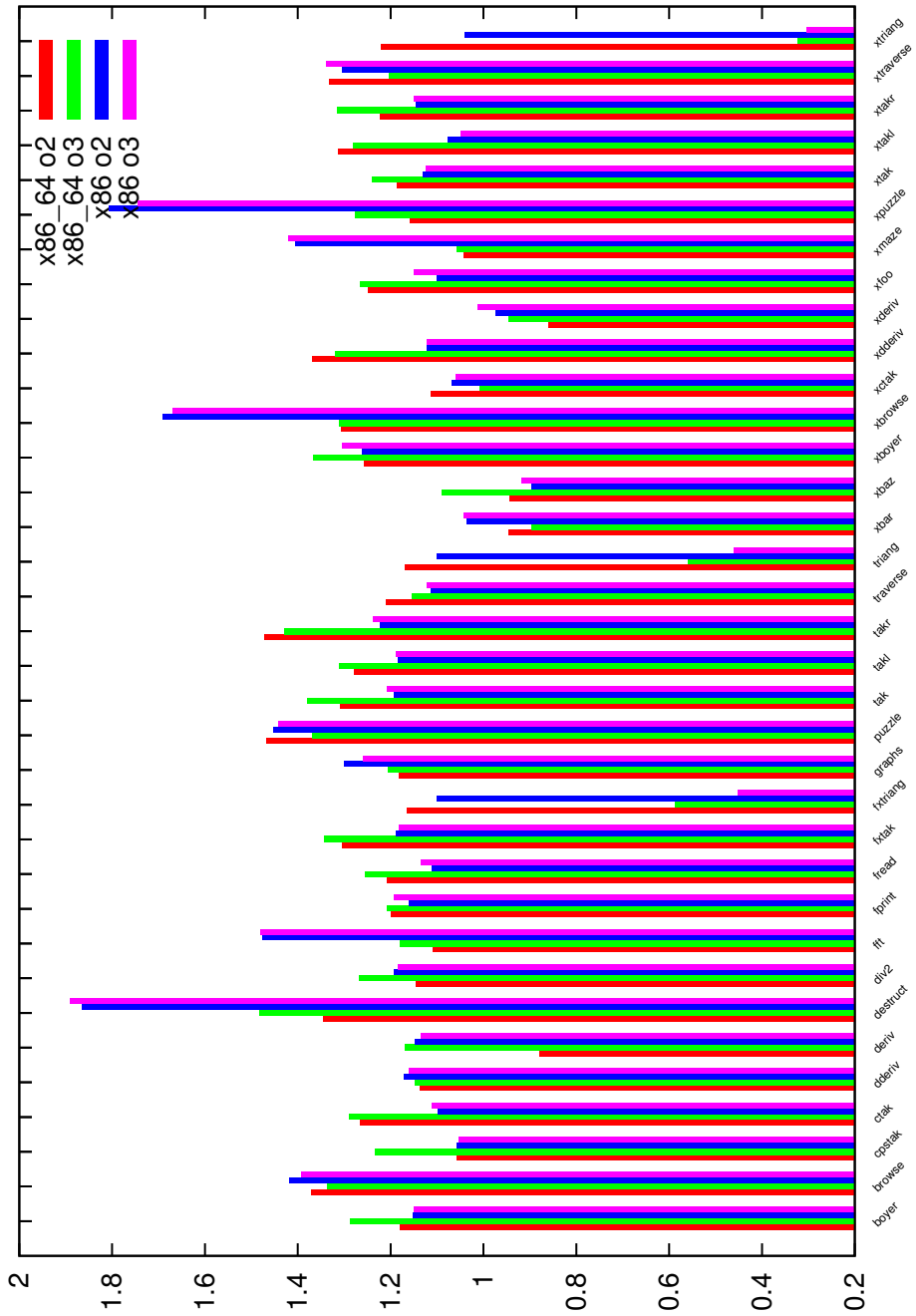

FIGURE 3.6. Normalized run time of `cp0` pass for the Chez Scheme benchmarks.

On the 64-bit version of Chez Scheme, the average normalized time is 1.22 at optimize level 2 and 1.21 at optimize level 3. This indicates, again, that there is some expense in the nanopass version of the pass. Again, however, by normalizing the total numbers, we can see that the total time is closer to being on par, with the run time of the pass being slower at a factor of 1.15 at optimize level 2 and a factor of 1.07 at optimize level 3. Again, the fine-grain timer is used to allow us to see details of the run times, which are often less than a millisecond.

3.5. Comparing the Speed of Generated Code

We set out with the goal of the new compiler generating code that is on par with the original compiler. We compare the performance of the original and new compiler on a set of benchmarks that includes the R6RS benchmarks [**26**]; a set of benchmarks, including some larger benchmarks, used to test the source optimizer; and a set of smaller benchmarks used for benchmarking Chez Scheme.

The benchmark data was generated on the same Intel® Core™ i7-3960X with two CPUs and six cores per CPU and 64 GB of RAM used for the individual pass timings. The tests were conducted at both optimize level 2 (run-time type-checking enabled) and optimize level 3 (run-time type-checking disabled), using both the 32-bit and 64-bit instruction sets. On average, the new compiler generates faster code, between 15.0% faster at optimize level 3, using the 64-bit instruction set, and 26.6% at optimize level 2, using the 32-bit instruction set.

Two of the benchmarks, `similix`, a self-applicable partial evaluator, and `softscheme`, a benchmark that performs soft typing, make use of the compiler during the run of the application. This negatively affects the run time of these benchmarks and the overall average, as the new compiler is slower than the original compiler. Table 3.5 shows the normalized run time of these benchmarks on both the 32-bit and 64-bit versions of the compiler and at optimize level 2 and optimize level 3.

TABLE 3.5. Normalized run times of the `similix` and `softscheme` benchmarks.

Optimization level	Machine type	Similix	Soft Scheme
2	x86	1.54	1.27
2	x86_64	1.41	1.32
3	x86	1.29	1.22
3	x86_64	1.39	1.24

Outside of these two outliers, the performance ranges on the 32-bit version between 0.417 and 1.05 at optimize level 2 and between 0.457 and 1.05 at optimize level 3 and, on the 64-bit version, between 0.519 and 1.03 at optimize level 2 and between 0.489 and 1.14 at optimize level 3. Figure 3.7 shows

the normalized performance of the R6RS benchmarks; Figure 3.8 shows the normalized performance of the benchmarks used for testing the source optimizer; and Figure 3.9 shows the normalized performance of the Chez Scheme benchmarks.

On average, benchmarks compiled with the new compiler outperform those compiled with the original compiler. As discussed in Section 3.3, the new compiler contains several improvements over the original compiler, and each of these contributes to the better performance of the benchmarks. The biggest contributing factor, and the one that is consistent in all of the benchmarks, is the graph-coloring register allocator.

The graph-coloring register allocator makes more registers available to the register allocator. In the original compiler, register allocation is performed on a set of higher-level operations. The assembler is then responsible for converting these higher-level operations into machine-specific assembly language, and it reserves registers for its use. This is particularly constraining on the 32-bit Intel® architecture, where only eight registers are available. Several of these registers are already spoken for as a means to store items such as the architectural stack pointer, the frame pointer, and the thread context. The new compiler exposes a set of operations that are much closer to machine-specific assembly language and exposes all of the operands to the register allocator.

3.6. Comparing Compilation Speed

We set out with the goal of implementing a new compiler that ran within a factor of two of the original compiler. The extra compile-time budget allows features such as a graph-coloring register allocator to be implemented in the new compiler, even though it is more expensive than the register allocator in the original compiler.

The two compilers are tested by compiling each benchmark five times and averaging the compilation times. The compile times are then normalized, using the original compiler as a base. On average, the compile times are better than the target times, averaging a factor of 1.71 on the 32-bit version at optimize level 2, 1.64 on the 32-bit version at optimize level 3, 1.75 on the 64-bit version at optimize level 2, and 1.71 on the 64-bit version at optimize level 3. These numbers include the time spent in garbage collection. If the garbage collection time is removed, the factor is 1.67 on the 32-bit version at optimize level 2, 1.60 on the 32-bit version at optimize level 3, 1.69 on the 64-bit version at optimize level 2, and 1.65 on the 64-bit version at optimize level 3. These numbers fall well within the goal that we set for ourselves. Nevertheless, further tuning is still necessary to try to get the compile times for the new compiler even closer to that of the original compiler.

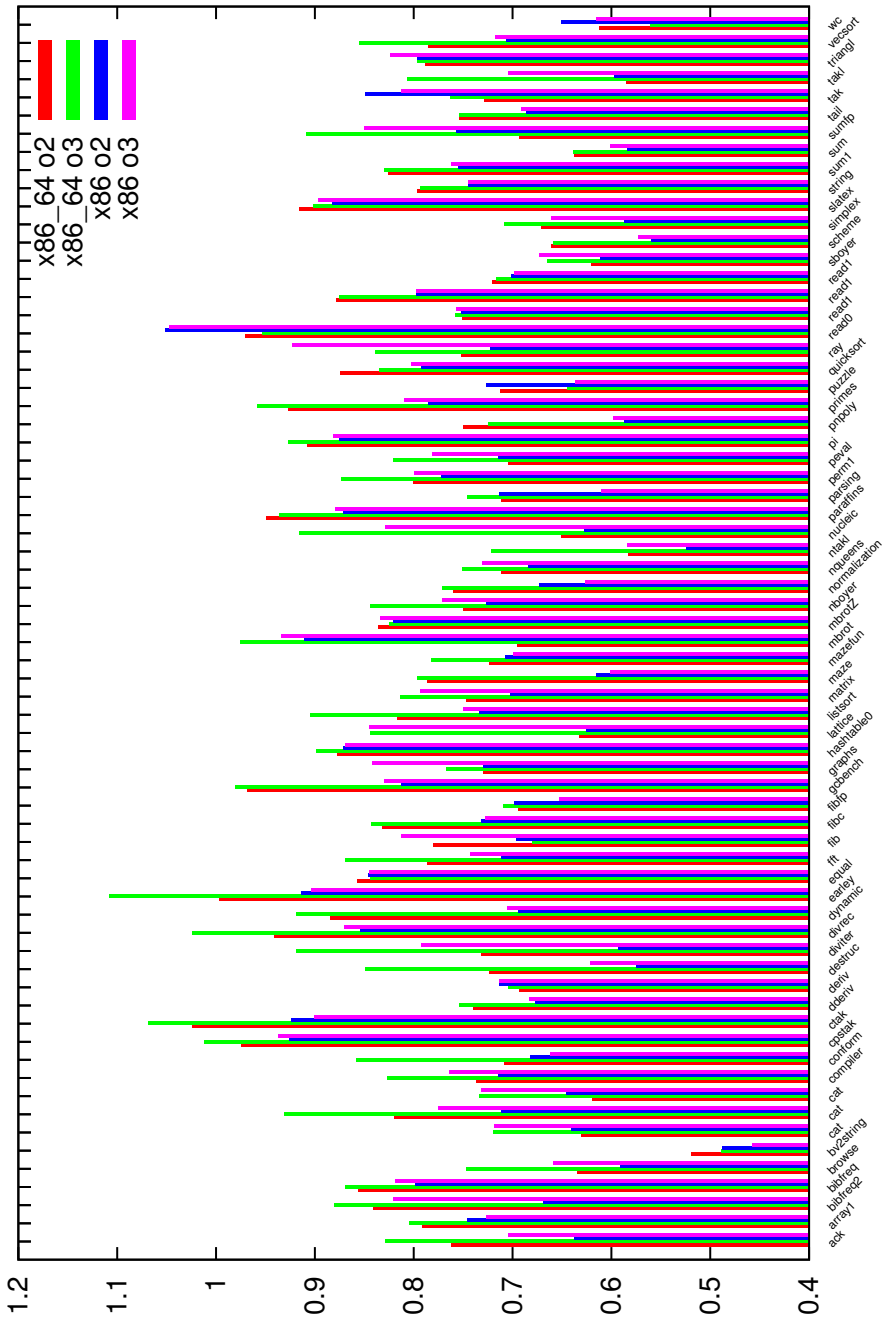

FIGURE 3.7. Normalized run-time performance of the R6RS benchmarks.

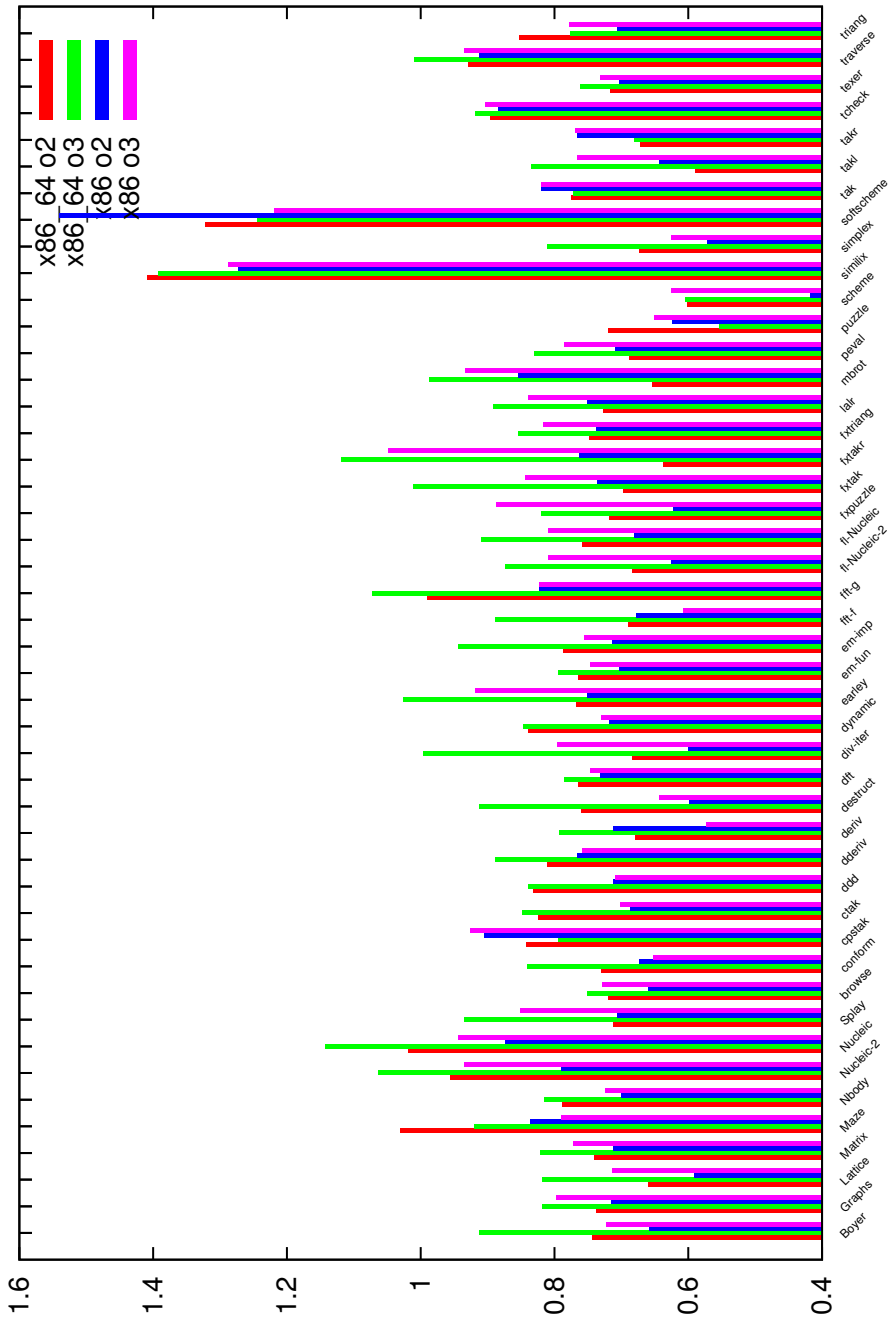

FIGURE 3.8. Normalized run-time performance of the source optimizer benchmarks.

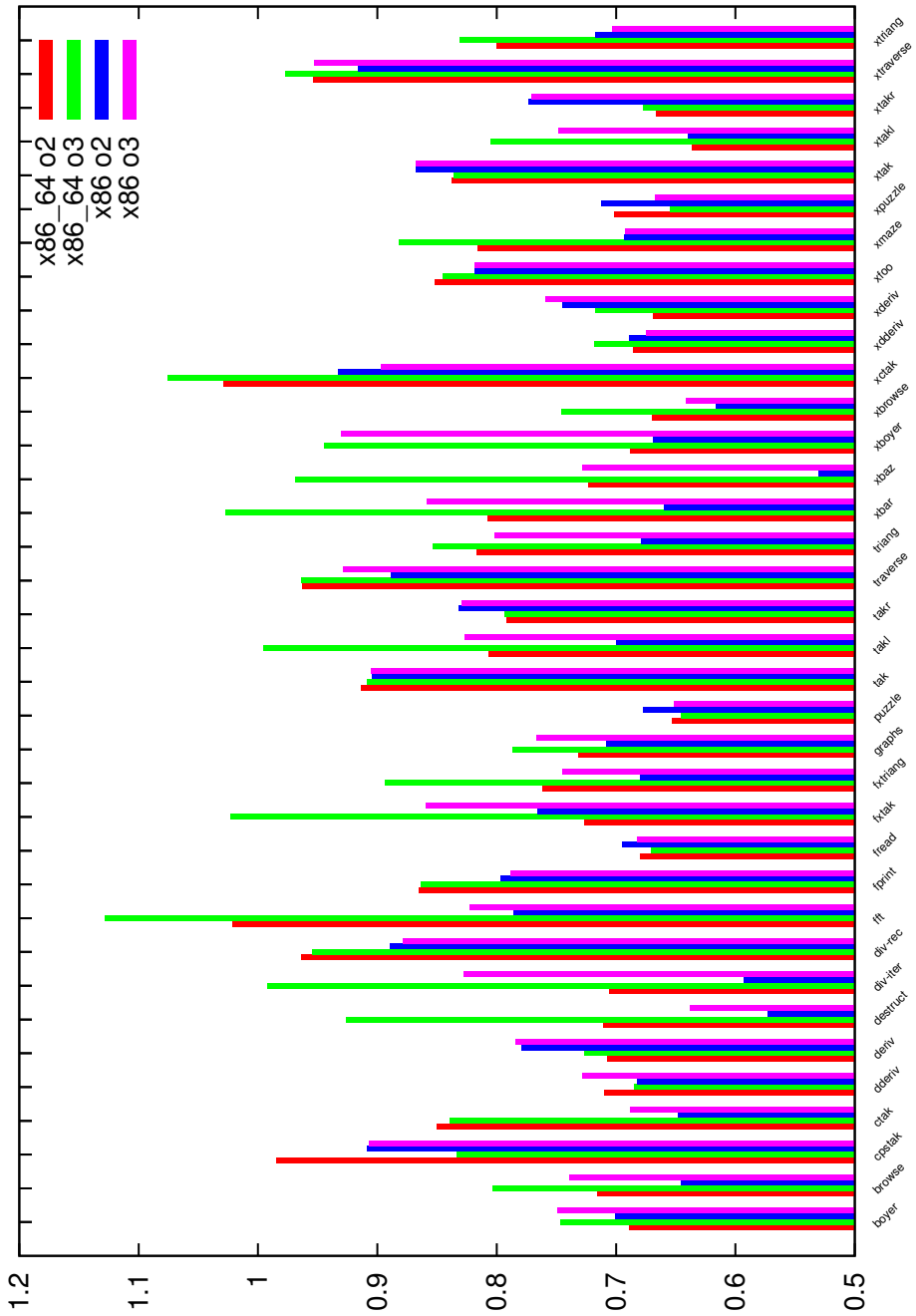

FIGURE 3.9. Normalized run-time performance of the Chez Scheme benchmarks.

The compile time varies from one benchmark to another. On the 32-bit version, the normalized time ranges from a factor of 1.00 to 4.44 at optimize level 2 and 0.968 to 3.13 at optimize level 3. On the 64-bit version, normalized time ranges from a factor of 1.00 to 4.73 at optimize level 2 and 1.00 to 3.80 at optimize level 3. Figure 3.10 shows the normalized compile times for the R6RS benchmarks; Figure 3.11 shows the compile times for the source optimizer benchmarks; and Figure 3.12 shows the normalized times for the Chez Scheme benchmarks.

It is natural to assume that the variance in compile times is related to the increased computational complexity of the graph-coloring register allocator; however, this does not seem to be the case. Figure 3.13 shows the normalized times versus the lines of expanded source code. The number of lines of expanded source code is determined by pretty printing the results of the expander run on each benchmark and then counting the number of line breaks. The graph in Figure 3.13 shows that some of the short benchmarks are also those that have the worst relative compile times. Overall, there does not seem to be a direct relationship between the number of lines of expanded code and the normalized compile time. An understanding of why this occurs might lead to overall improvements in compile time.

3.6.1. Breaking down the front-end and back-end. The front-ends of the two compilers are similar, with the same set of passes implementing the same algorithms. The only difference between the two is that the front-end of the new compiler uses the nanopass framework, where the original compiler uses a tagged vector representation. The back-ends of the compilers, in contrast, diverge significantly. Thus, it makes sense to break the compilation times down into front-end and back-end times to help determine the source of the overhead.

3.6.1.1. Comparing compiler front-ends. The front-ends of both compilers often run fast enough on the benchmarks that the fine-grain timer is needed to measure the time used. As expected, the time of the front-end for both compilers is similar. These times include the time taken by the Scheme reader, which is the same on both compilers. On average, the 32-bit version of the compiler had a compile time at a relative factor of 0.96 at both optimize level 2 and optimize level 3. The 64-bit version of the compiler had a compile time at a relative factor of 1.03 at both optimize level 2 and optimize level 3. The similar times on the front-end of the two compilers demonstrates that the nanopass framework adds only minimal overhead when compared with the tagged vector representation used by the original compiler.

Figure 3.14 shows the normalized times for running the front-end of the compiler with the R6RS benchmarks; Figure 3.15 shows the normalized times with the source optimizer benchmarks; and

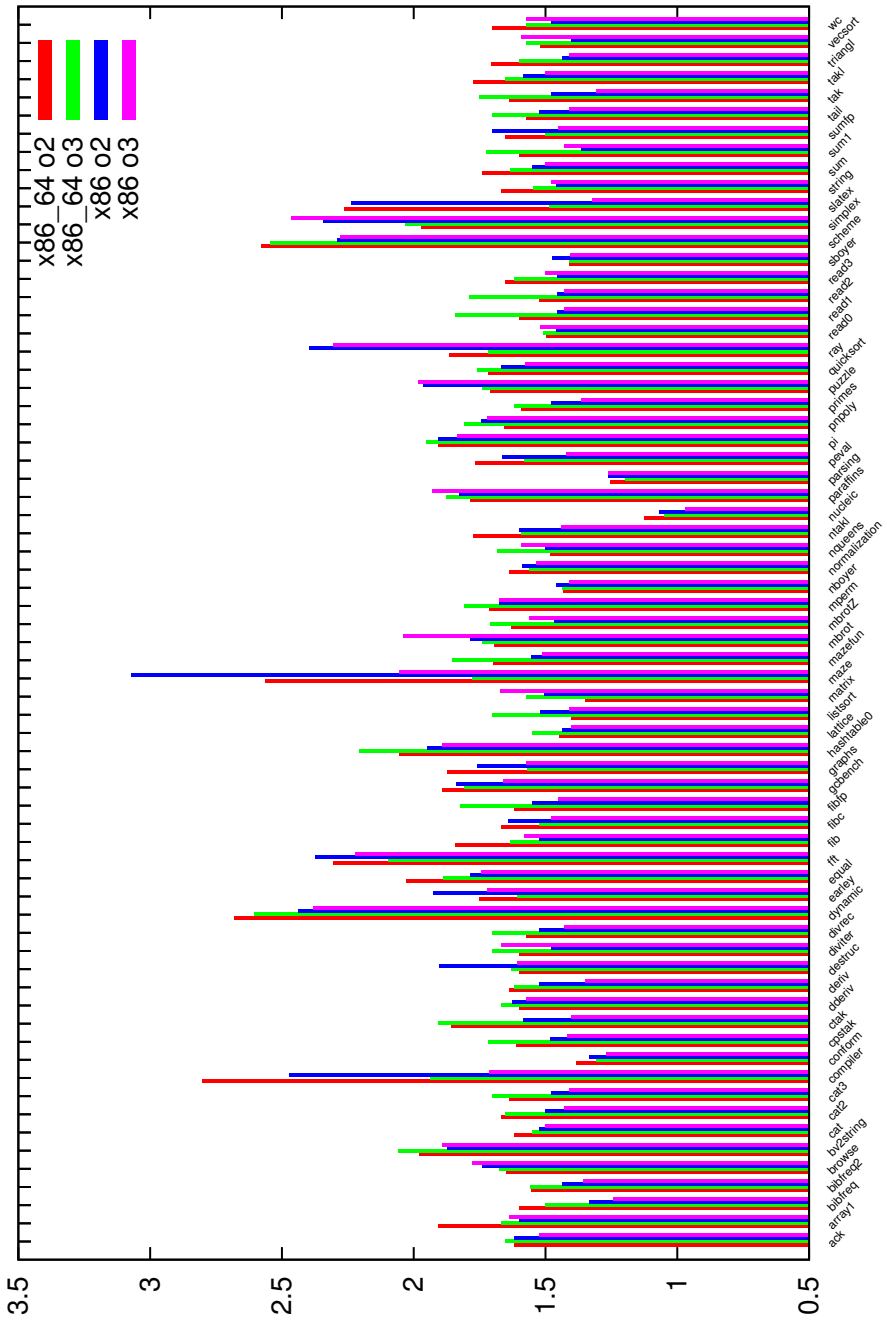

FIGURE 3.10. Normalized compile-time performance of the R6RS benchmarks.

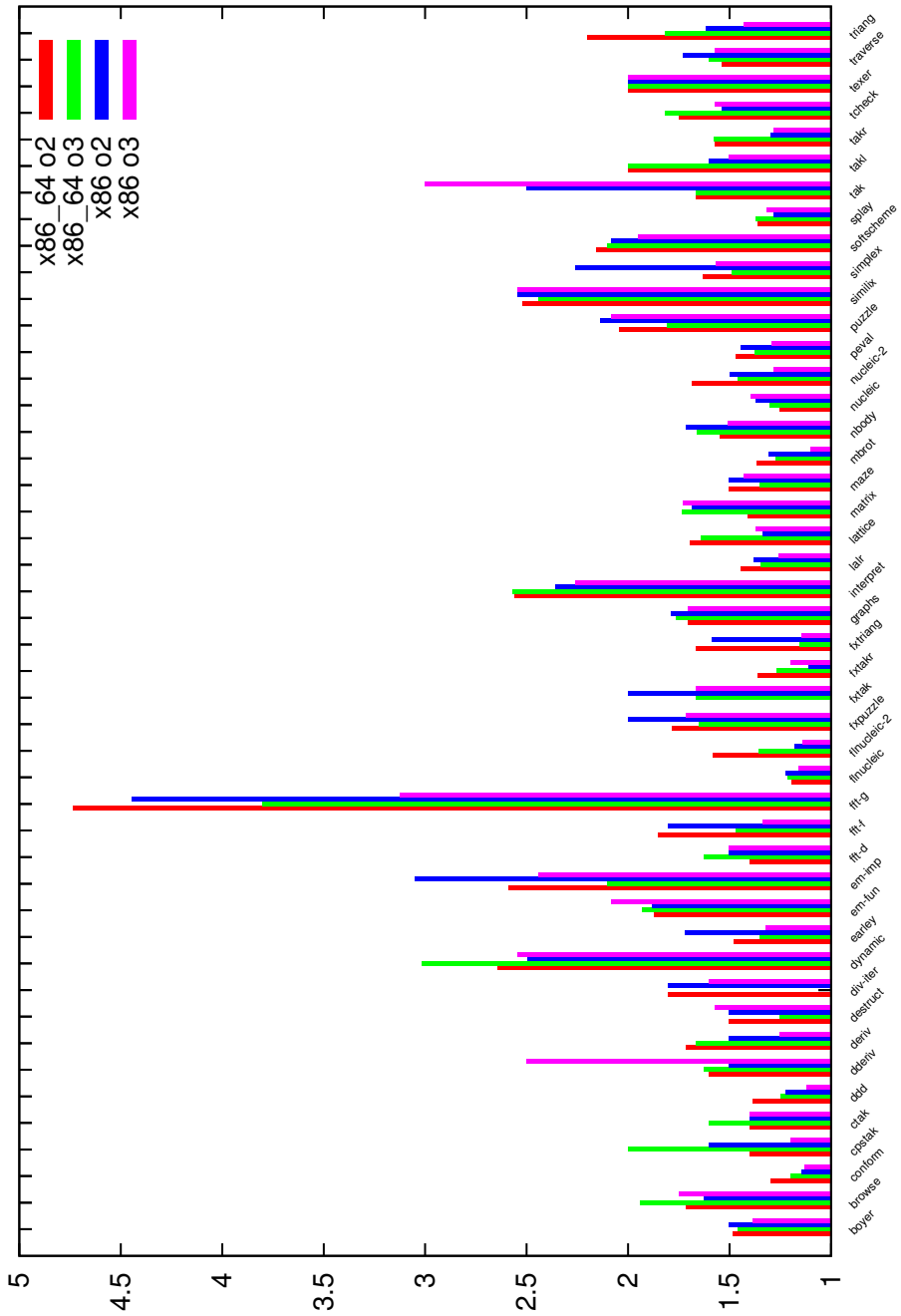

FIGURE 3.11. Normalized compile-time performance of the source optimizer benchmarks.

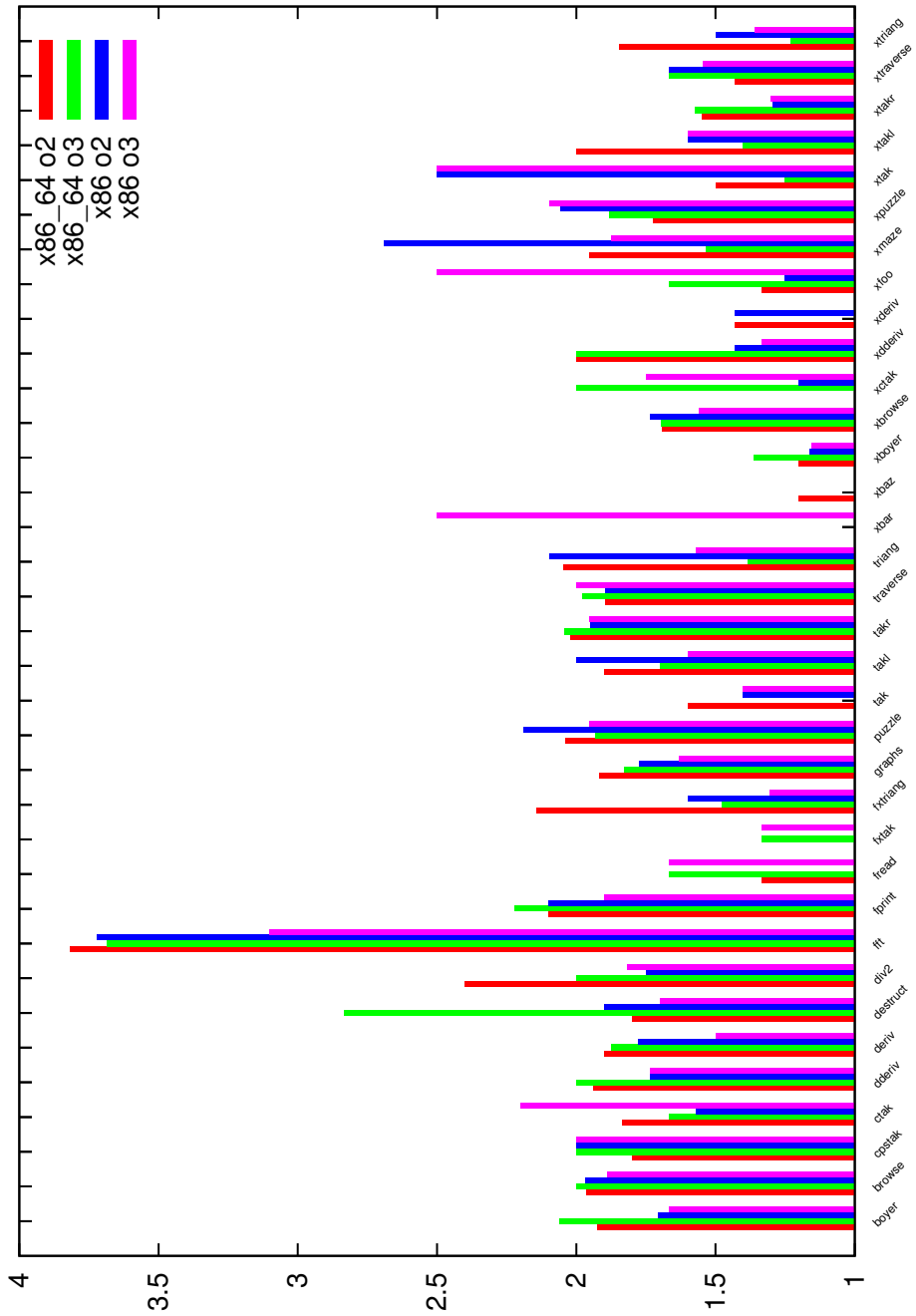

FIGURE 3.12. Normalized compile-time performance of the Chez Scheme benchmarks.

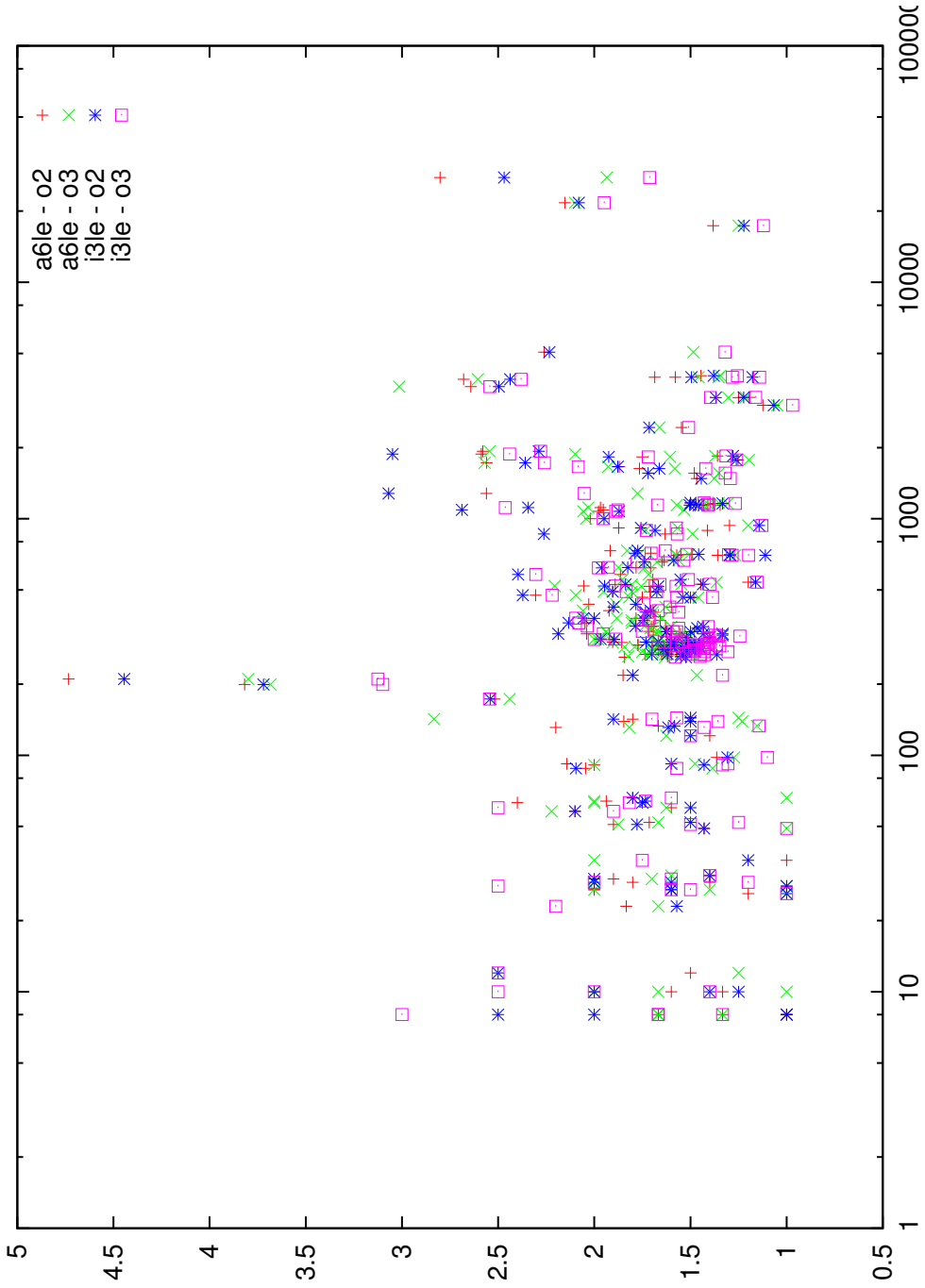

FIGURE 3.13. Normalized compile times vs. the size of the expanded source code.

Figure 3.16 shows the normalized front-end times for the Chez Scheme benchmarks. The normalized average on the 32-bit version of the compilers ranges from 0.736 to 1.71 at optimize level 2 and 0.455 to 1.54 at optimize level 3. The normalized average on the 64-bit version of the compilers ranges from 0.835 to 1.73 at optimize level 2 and 0.476 and 1.63 at optimize level 3. The front-end spends more time in the garbage collector for the `softscheme` benchmark on the new compiler, which is part of the reason that it is at the top of the normalized times.

3.6.1.2. Comparing compiler back-ends. Unlike the front-ends of the two compilers, the back-ends vary significantly and contribute to most of the compile-time difference. The 32-bit version of the new back-end runs at an average normalized time of 2.01 at optimize level 2 and 1.91 at optimize level 3. The 64-bit version of the new back-end runs at an average normalized time of 2.14 at optimize level 2 and 2.11 at optimize level 3. These times are based on the fine-grain timer, as some of the benchmarks compile fast enough that there is no measurable time at the granularity of the normal Chez Scheme statistics package. Figure 3.17 shows the normalized times for running the back-end of the compiler with the R6RS benchmarks; Figure 3.18 shows the normalized times with the source optimizer benchmarks; and Figure 3.19 shows the normalized back-end times for the Chez Scheme benchmarks.

3.6.2. Analyzing pass overhead. Although the new compiler performs well, despite having ten times as many passes in the back-end, we wanted to get a better understanding of the expense of adding new passes. This expense is difficult to measure directly, partially because the new compiler has several "passes" that avoid traversing the entire intermediate representation of the program and partially because the number of nodes in the intermediate representation increase as primitives are expanded into inline code and machine-specific constraints are imposed on the results.

To try to measure this we have devised two measurement methods. The first adds a new "dummy" pass for each real pass and times the running of both the dummy pass and the real pass on the input. The traversal overhead for the pass is then calculated by dividing the time for the dummy pass by the time for the real pass. These are averaged for all of the passes in the compiler to get an idea of the aggregate overhead.

The second adds only three dummy passes in strategic places in the compiler to determine the cost of adding a pass near the entry point of the back-end, after primitive expansion, and after machine-specific constraints are imposed. These passes are timed, along with the real passes of the compiler, and their effect is determined by dividing the timing for each by the total run time for the back-end, excluding the dummy passes.

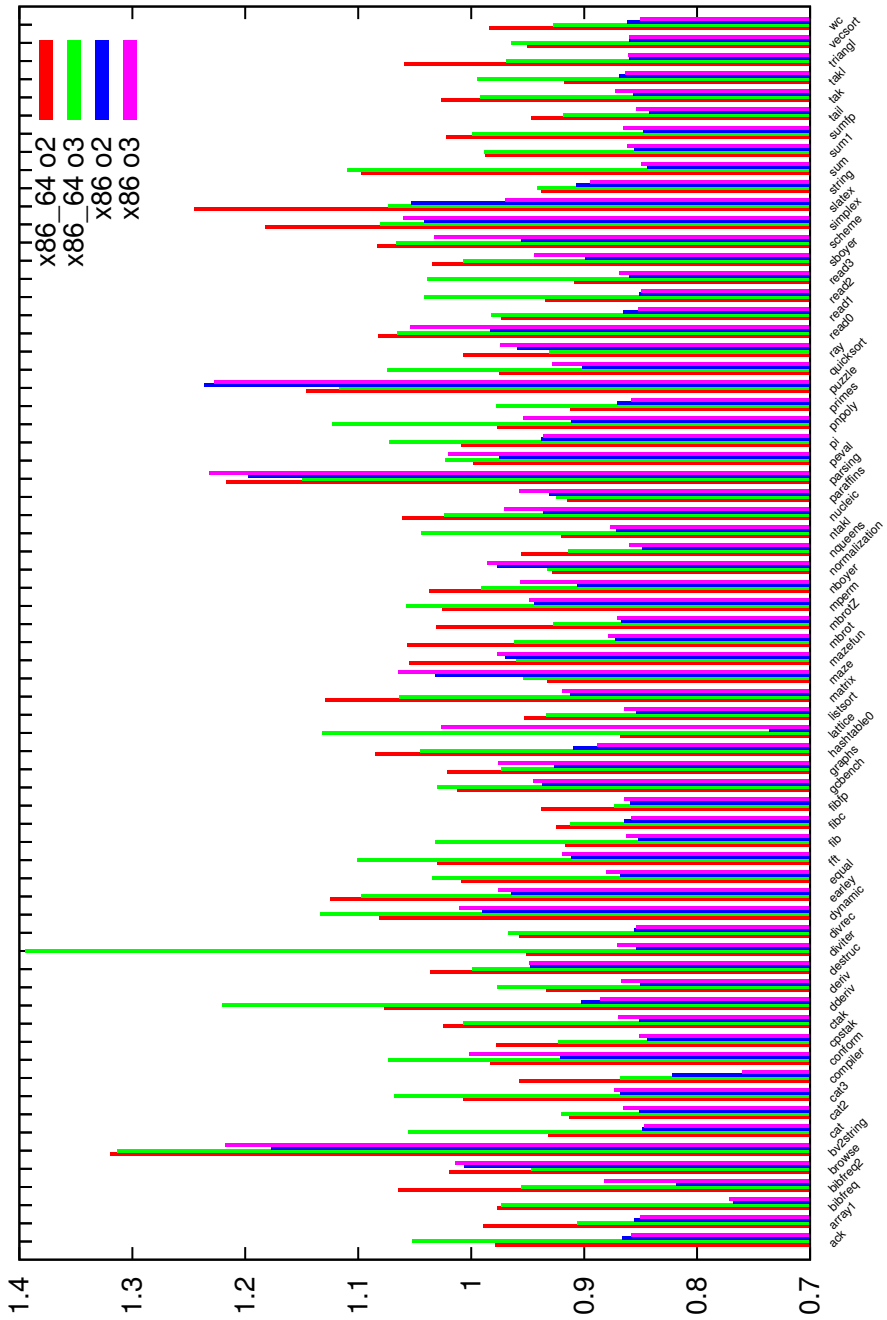

FIGURE 3.14. Normalized front-end compile-time performance of the R6RS benchmarks.

3.6.2.1. Traversal overhead. To determine an estimate of the average traversal overhead for each pass we autogenerated a dummy pass for each real pass and timed the running of each pass with the same input. The new compiler uses a handful of common syntactic forms that allows us to instrument each pass in a similar way to record timing statistics and to inspect the output of passes by name when the compiler is running. This provided a hook for the dummy passes to be added. Unfortunately, not every pass in the compiler traverses the entire intermediate program term, and some of those that do, do not generate an output term. In fact, some passes do not look at the intermediate program term at all because they are driven entirely by information gathered outside the program term. Each dummy pass, however, always traverses the entire intermediate program term and produces a new output term. This mismatch in effort skewed the results, as a dummy pass might take longer to run than a real pass that does not need to traverse the entire term.

To avoid this problem, we did not include in the average traversal overhead those passes that do not traverse the entire program term and those that do not produce an output term. This excludes a number of passes in the register allocator that either operate over the list of variables needing frame or register allocation or that operate only over information captured by the basic blocks of the program term and do not traverse the contents of the basic block.

Even with this provision in place, the garbage collector sometimes causes a pass to take more time than it otherwise would. To avoid the affects of the garbage collector, we removed the time spend in garbage collection from the time for each pass. Both the pass time and garbage collection time are measured using the fine-grain timer, as the pass times are often too small to measure with the millisecond statistics package.

The same three suites of benchmarks used throughout this chapter are also used to determine the traversal overhead. On the 32-bit version of the compiler, the traversal overhead is 68.5% at optimize level 2 and 68.6% at optimize level 3. On the 64-bit version of the compiler, the traversal overhead is 68.8% at optimize level 2 and 69.0% at optimize level 3.

The traversal overhead on the 32-bit version ranges from 62.0% to 81.8% at optimize level 2 and 61.9% to 82.4% at optimize level 3. The traversal overhead on the 64-bit version ranges from 63.6% to 80.8% at optimize level 2 and 62.1% to 83.2% at optimize level 3. These numbers appear to indicate that the majority of the time in each pass is spent traversing the input-language term and constructing the output-language term. Because many of the passes measured change only a few productions from the input-language term, it makes sense that the timing for each dummy pass and the related pass are similar.

3.6.2.2. Overhead at each expansion stage. Although the traversal overhead gives us some insight into the time of a dummy pass relative to its equivalent real pass, it does not inform us directly about the expense of adding a pass to a given place in the compiler. To understand this better, we created three dummy passes, one for each step in the compiler where the code expands. The first dummy pass is inserted immediately following the first back-end pass. This helps determine the expense of adding a pass before any primitive expansion has occurred. We will refer to this pass as the source pass. The second dummy pass is inserted immediately following the primitive expansion pass. We will refer to this pass as the post-primitive expansion pass. The final dummy pass is inserted immediately following the instruction selector, which imposes machine-specific constraints and produces an output-language term that is close to assembly language. We will refer to this pass as the post-instruction selection pass.

Table 3.6 shows the percentages of the back-end compile time spent in each of the passes on 32-bit and 64-bit versions of the compiler at optimize level 2 and optimize level 3. The numbers in the table reflect our natural intuition that the traversal expense of a pass increases as the program term that is being compiled expands. This suggests that, if we could find a way to avoid expansion until later in the compiler, this could improve performance, even without reducing the total number of passes. The original compiler, for instance, avoids most of this expansion until its final pass. The new compiler could also, potentially, avoid some of this expansion by using a higher-level set of operations, instead of using low-level instructions similar to assembly language. Unfortunately, doing this might mean sacrificing some of the benefits of the graph-coloring register allocator.

TABLE 3.6. Percentage of back-end time for each of three dummy passes, one after the initial back-end pass, one after the primitive expansion pass, and one after the instruction selection pass.

Opt. level	Machine	Source	Post-primitive exp.	Post-instruction select.
2	x86	0.198%	0.529%	2.19%
2	x86_64	0.217%	0.624%	2.19%
3	x86	0.236%	0.527%	2.18%
3	x86_64	0.258%	0.602%	2.20%

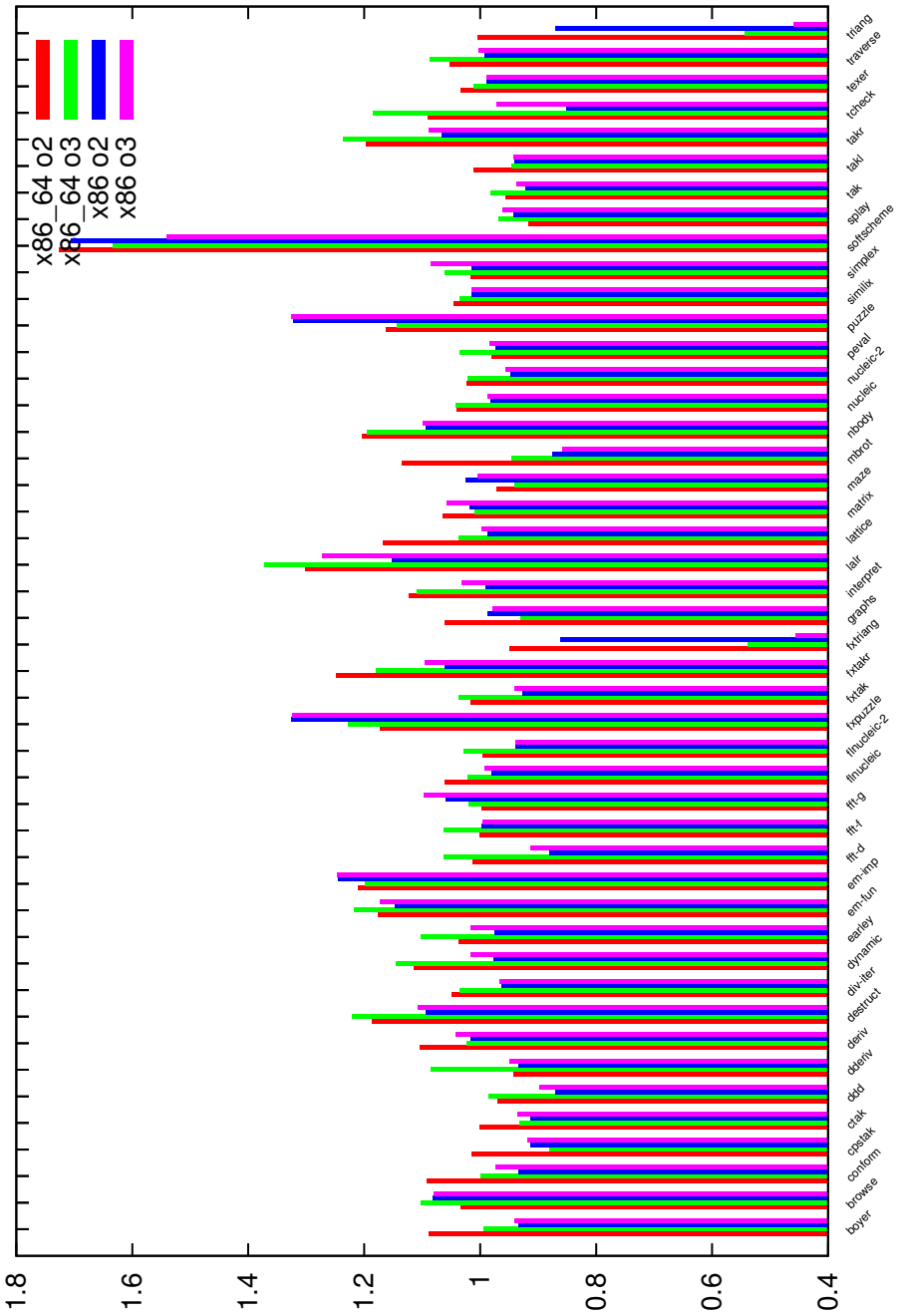

FIGURE 3.15. Normalized front-end compile-time performance of the source optimizer benchmarks.

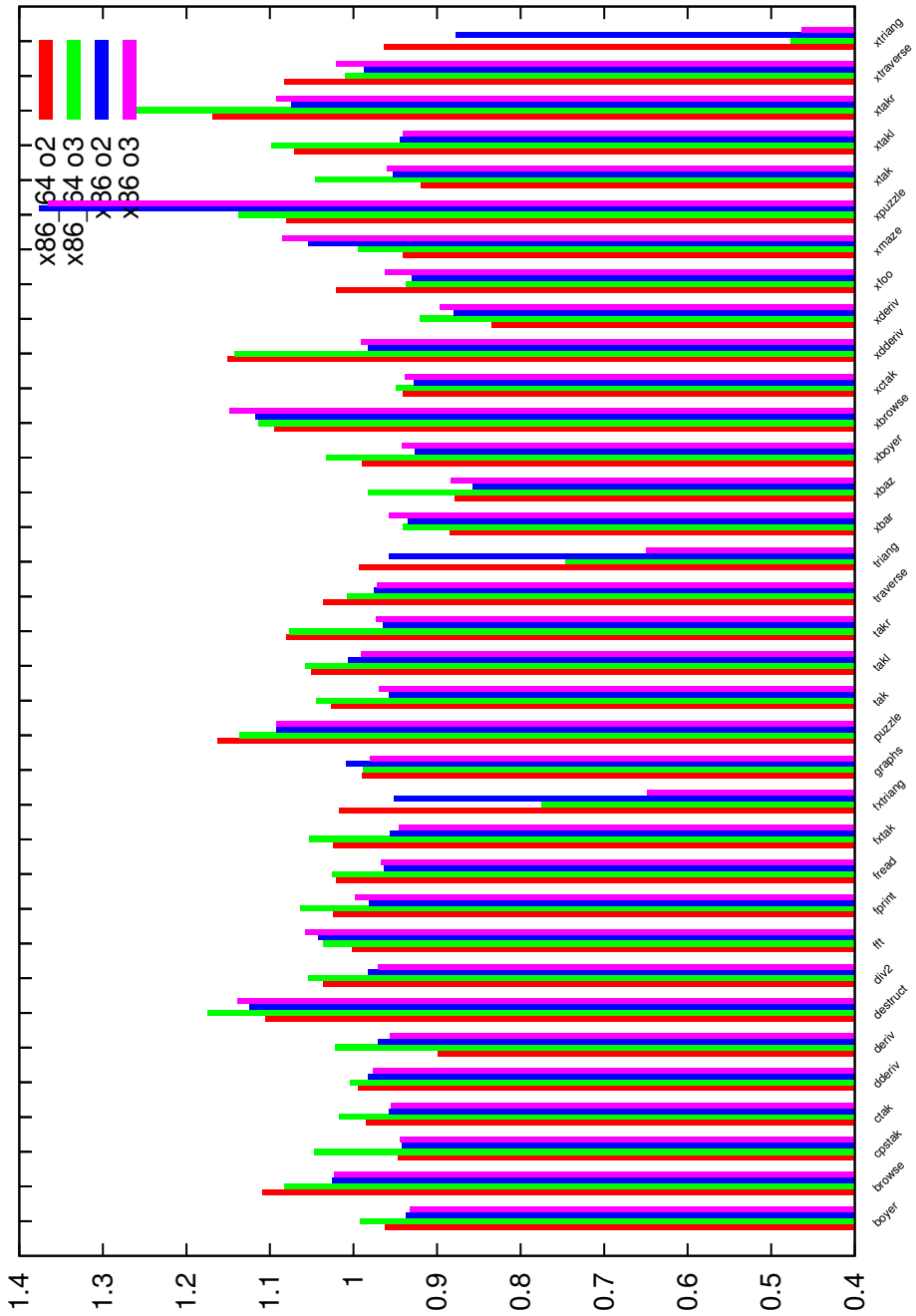

FIGURE 3.16. Normalized front-end compile-time performance of the Chez Scheme benchmarks.

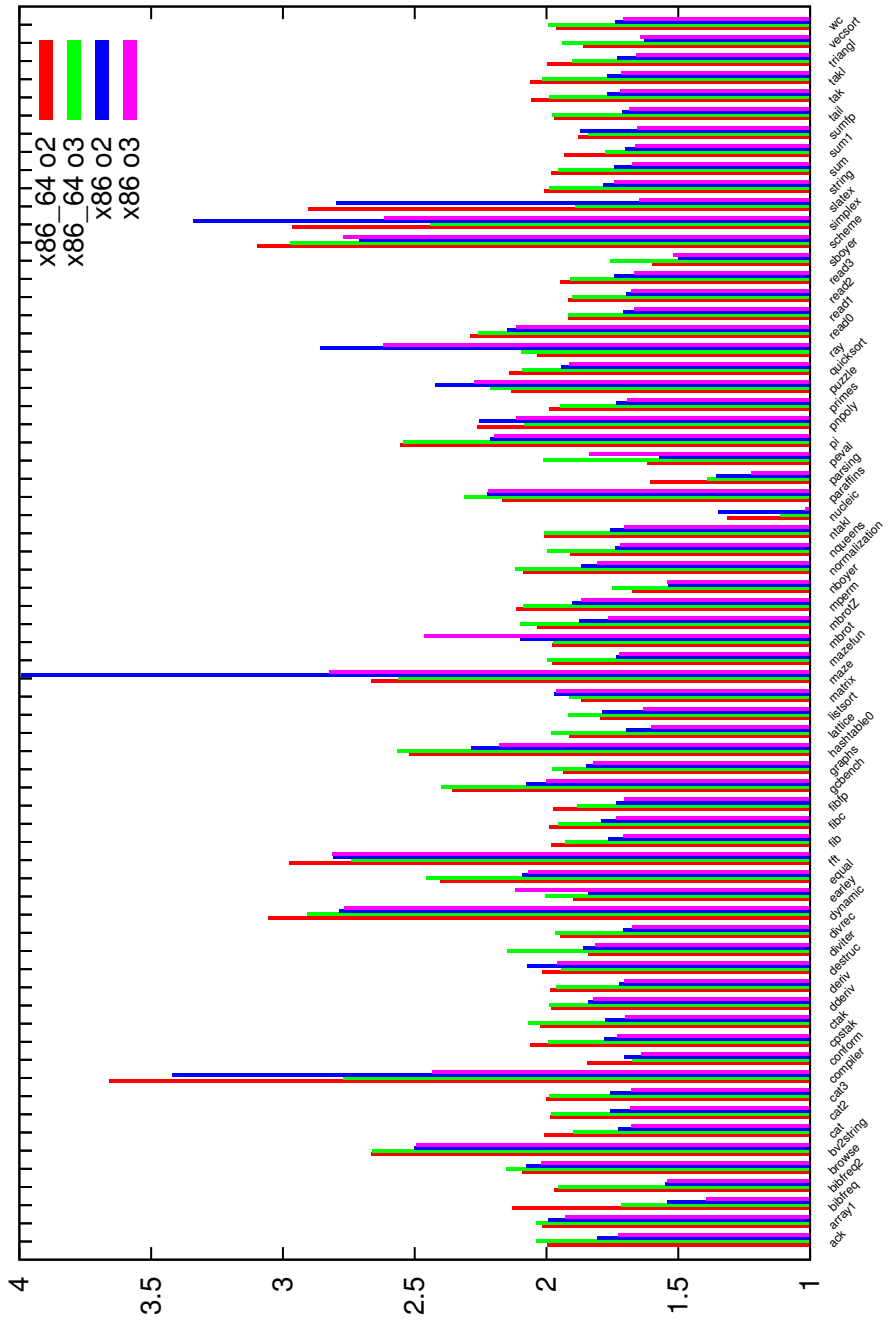

FIGURE 3.17. Normalized back-end compile-time performance of the R6RS benchmarks.

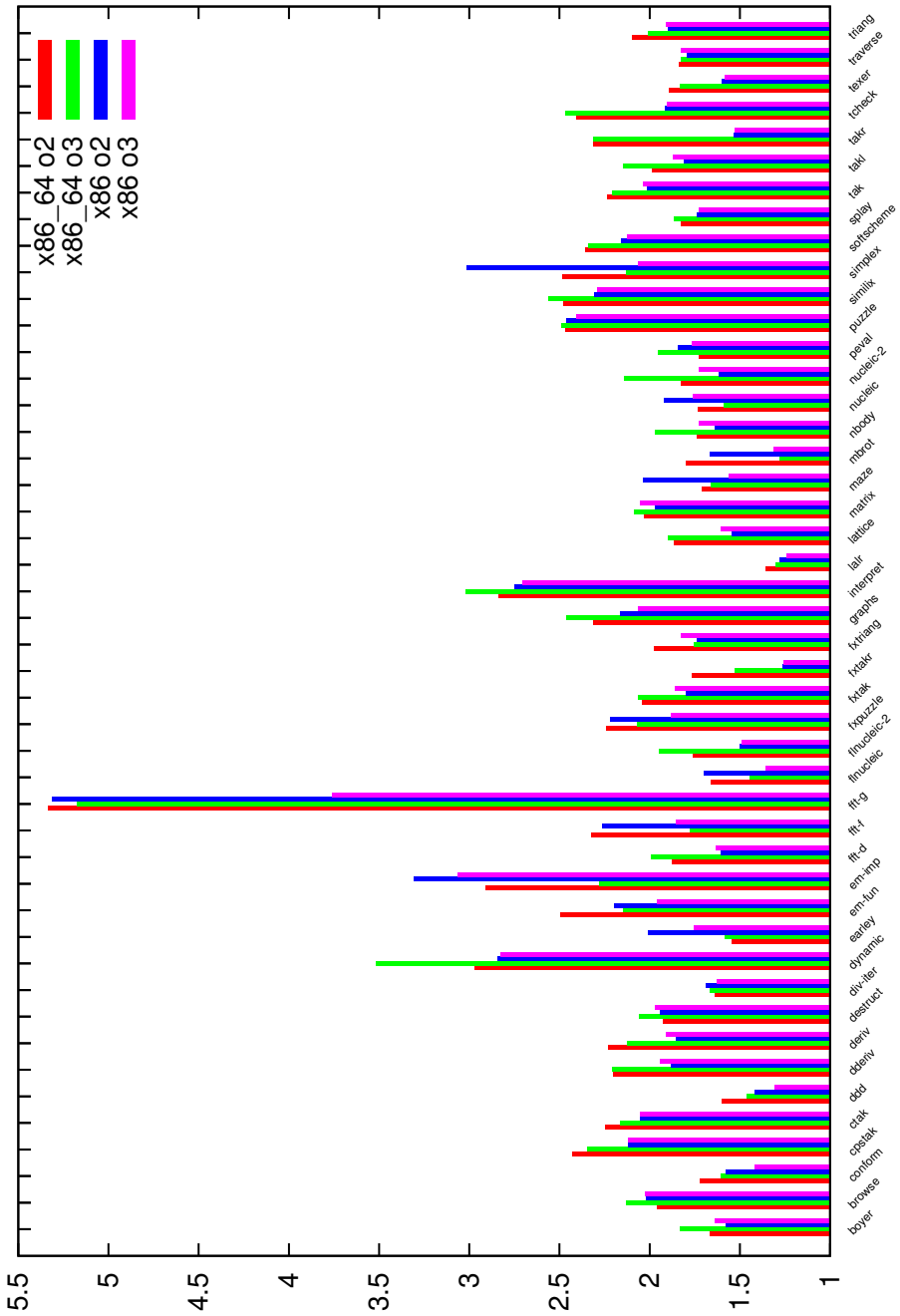

FIGURE 3.18. Normalized back-end compile-time performance of the source optimizer benchmarks.

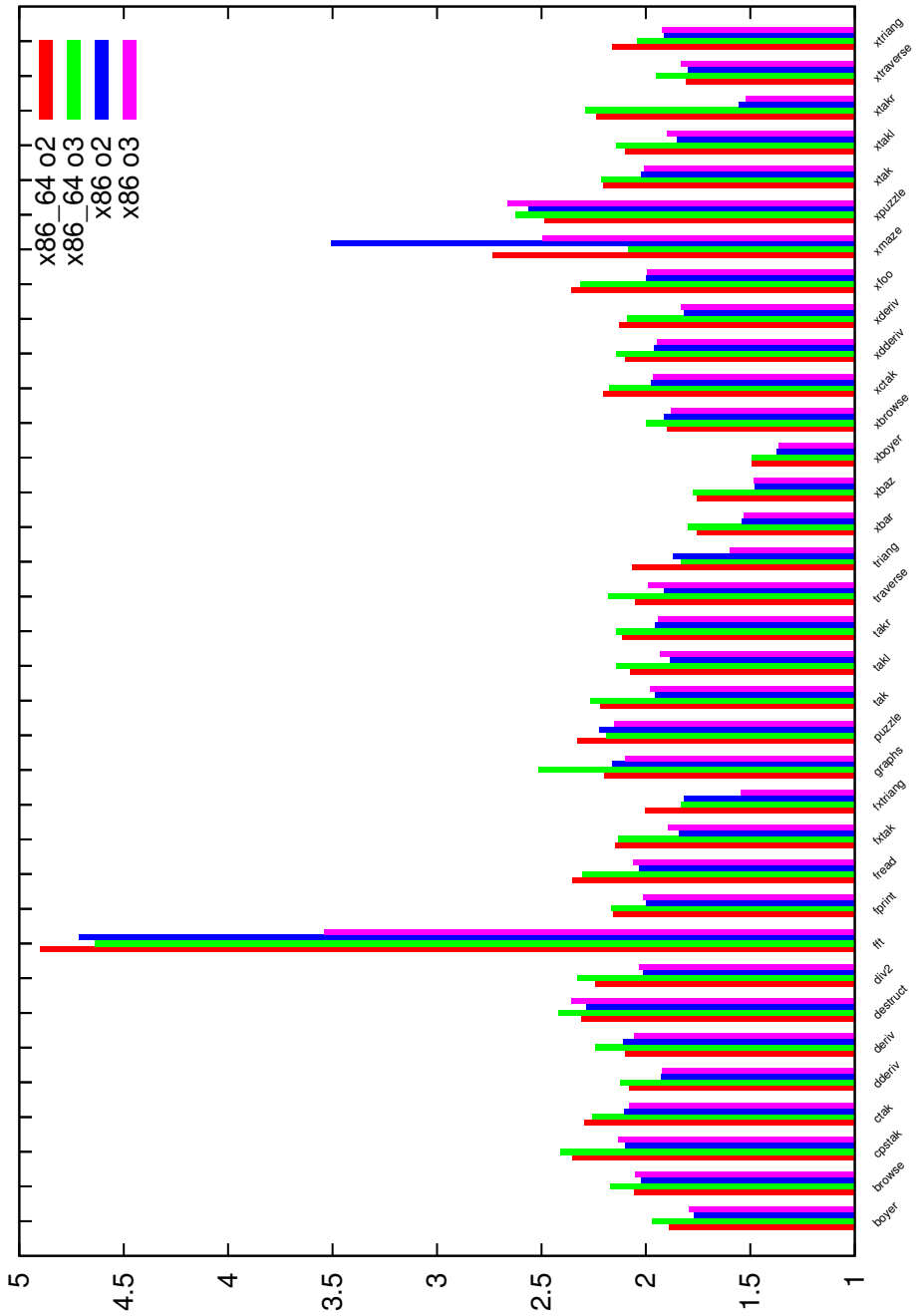

FIGURE 3.19. Normalized back-end compile-time performance of the Chez Scheme benchmarks.

CHAPTER 4

Cross-library Optimization

4.1. Introduction

A major difference between the language defined by the Revised[6] Report on Scheme (R6RS) and earlier dialects is the structuring of the language into a set of standard libraries and the provision for programmers to define new libraries of their own [**83**]. New libraries are defined via a `library` form that explicitly names its imports and exports. No identifier is visible within a library unless explicitly imported into or defined within the library. As such, each library essentially has a closed scope that, in particular, does not depend on an ever-changing top-level environment, as in earlier Scheme dialects. Further, the exports of a library are immutable, both in the exporting and importing libraries. The compiler (and programmer) can thus be certain that, if `cdr` is imported from the standard base library, it really is `cdr` and not a variable whose value might change at run time. This is a boon for compiler optimization, as it means that `cdr` can be open coded or even folded, if its arguments are constants.

Another boon for optimization is that procedures defined in a library, whether exported or not, can be inlined into other procedures within the library, as there is no concern that some importer of the library can modify the value. For the procedures that a compiler cannot or chooses not to inline, the compiler can avoid constructing and passing unneeded closures, bypass argument-count checks, branch directly to the proper entry point in a `case-lambda`, and perform other related optimizations [**33**].

Yet another benefit of the closed scope and immutable bindings is that the compiler often can recognize most or all calls to a procedure from within the library in which it is defined and verify that an appropriate number of arguments is being passed to the procedure; it can also issue warnings when it determines that this is not the case. If the compiler performs some form of type recovery [**81**], it might also be able to verify that the types of the arguments are correct, despite the fact that Scheme is a latently typed language.

The success of the library form can be seen by the number of libraries that are already available [**2**]. Part of the success can be traced to the portable library implementations produced by Van Tonder [**88**] and Ghuloum and Dybvig [**55**]. The portable library implementations form the basis for at least two R6RS Scheme implementations [**25**, **51**], and Ghuloum's system is available on a variety of R5RS Scheme implementations [**52**].

The library mechanism is specifically designed to allow separate compilation of libraries, although it is generally necessary to compile each library upon which a library depends before compiling the library itself [**53**, **55**]. Thus, it is natural to view each library as a single compilation unit, which is what existing implementations support. Yet, separate compilation does not directly support three important features:

- cross-library optimization, e.g., inlining, copy propagation, lambda lifting, closure optimizations, type specialization, and partial redundancy elimination;
- extension of static argument count and type checking across library boundaries; and
- the merging of multiple libraries (and possibly an application's main routine) into a single object file so that the distributed program is self-contained and does not expose details of the structure of the implementation.

This chapter introduces two methods to support these features. The first is the `library-group` form that allows a programmer to specify a set of libraries and an optional program to be combined as a single compilation unit. The second is implicit cross-library optimization, where an intermediate representation of some constants and procedures are attached to the identifiers exported from a library and can be inlined when the identifier is referenced in another library or program.

Each library contained within the group might or might not depend on other libraries in the group, and if an application program is also contained within the group, it might or might not depend on all of the libraries. In particular, additional libraries might be included for possible use (via `eval`) when the application is run. The `library-group` form does not require the programmer to restructure the code. That is, the programmer can continue to treat libraries and programs as separate entities, typically contained in separate source files, and the libraries and programs remain portable to systems that do not support the `library-group` form. The `library-group` form merely serves as a wrapper that groups existing libraries together for purposes of analysis and optimization but has no other visible effect. Even though the libraries are combined into a single object file, each remains visible separately outside of the group.

For many languages, the `library-group` form would be almost trivial to implement. In Scheme, however, the implementation is complicated significantly by the fact that the compilation of one library can involve the use of another library's run-time bindings. That is, as each library in a library group is compiled, it might require another in the same group to be compiled and loaded. This need arises from Scheme's procedural macros. Macros are defined by transformers that are themselves coded in Scheme. Macro uses are expanded at compile time or, more precisely, *expansion time*, which precedes compilation. If a macro used in one library depends on the run-time bindings of another, the other must be loaded before the first library can be compiled. This need arises even when libraries do not export keyword (macro) bindings, although the export of keywords can cause additional complications.

As with libraries themselves, the `library-group` implementation is handled entirely by the macro expander and adds no burdens or constraints on the rest of the compiler. This makes it readily adaptable to other implementations of Scheme and even to implementations of other languages that support procedural macros, now or in the future.

The `library-group` form provides a powerful mechanism for combining libraries, but we would also like to provide automatic cross-library optimization when the `library-group` form is not used. In this case, when a library is compiled to a file, extra information is recorded in the binary that allows cross-library optimization to occur when the library is used. Rather than storing a near-source internal representation for each exported run-time identifier, the source optimization pass [**90**] helps identify constants and procedures that are likely to be *inlinable*. When another library or program imports the compiled library, identifiers with this information attached participate in the source optimization process as if the source code were included in the same source file. Not all constants can be copied, however, as constant pairs, vectors, and other quoted complex data types must remain pointer-equivalent, and copying breaks this equivalence. Procedures are considered *inlinable* if they are both small enough to be inlined in the library in which they are defined and have no free variables that refer to local bindings within the library.[4]

Even with the addition of implicit cross-library optimizations, the `library-group` form is still necessary for obtaining certain kinds of optimizations. For instance, constants for constructing items such as pairs, vectors, and records that cannot be copied with implicit cross-library optimization can be constant propagated across library boundaries within the `library-group` form. Similarly, procedure code that was too large to be included in a cross-library optimization might still be

[4]All definitions in a library are considered local bindings to allow inlining within the library, but references to identifiers imported from other libraries or to primitives are not.

small enough to inline when applied directly in the body of another library or program with the `library-group` form. References to library globals, or to variables bound within the body of a library, that cannot be used by the implicit cross-library optimizations can potentially be inlined across library boundaries within the `library-group` form. This is possible because the library group has access to all of the source code of the defining library, even that which is not exported, and the dependencies of the library group inherently include the full dependencies for all of the libraries included in the library group.

While the `library-group` form is handled entirely in the expander, automatic cross-library optimization requires both the expander and the source optimization pass to be involved. When a library is compiled, the expander creates placeholders for the information to be associated with the exported identifier, and the source optimizer fills in this location when appropriate. When a library identifier that has inlining information is referenced in call position, the expander is responsible for replacing the reference with the inlinable expression, when it is available. The source inliner also replaces references to library globals. If the optimization information for a library global contains a quoted constant or primitive reference, it is always inlined. If the optimization information contains a procedure expression, it is inlined when it is in call context or is replaced with a true value when it is in test context.

The remainder of this chapter is organized as follows. Section 4.2 provides background on the `library` form and Ghuloum's library implementation, which we use as the basis for describing our implementation. Section 4.3 introduces the `library-group` form and contains a discussion of what the expander produces for a library group and how it does so. Section 4.4 presents our implementation of automatic cross-library optimization. Section 4.5 provides an illustration of when cross-library optimization is helpful. Sections 4.6 and 4.7 include related and future work, and Section 4.8 presents our conclusions.

4.2. Background

This section includes a description of R6RS libraries and top-level programs, which are the building blocks for our library groups. It also covers those aspects of Ghuloum's implementation of libraries that are relevant to our implementation of library groups.

4.2.1. Libraries and top-level programs. An R6RS library is defined via the `library` form, as illustrated by the following trivial library.

```
(library (A)
  (export fact)
  (import (rnrs))
  (define fact
    (lambda (n)
      (if (zero? n) 1 (* n (fact (- n 1)))))))
```

The library is named `(A)`, exports a binding for the identifier `fact`, and imports from the `(rnrs)` library. The `(rnrs)` library exports bindings for most of the identifiers defined by R6RS, including `define`, `lambda`, `if`, `zero?`, `*`, and `-`, which are used in the example. The body of the library consists only of the definition of the exported `fact`.

For our purposes,[5] library names are structured as lists of identifiers, e.g., `(A)`, `(rnrs)`, and `(rnrs io simple)`. The `import` form names one or more libraries. Together with the definitions in the library's body, the imported libraries determine the entire set of identifiers visible within the library's body. A library's body can contain both definitions and initialization expressions, with the definitions preceding the expressions. The identifiers defined within a library are either run-time variables, defined with `define`, or keywords, defined with `define-syntax`.

Exports are simply identifiers. An exported identifier can be defined within the library, or it can be imported into the library and re-exported. In Scheme, types are associated with values, not variables, so the `export` form does not include type information, as it typically would for a statically typed language. Exported identifiers are immutable. Library `import` forms cannot result in cyclic dependencies; thus, the direct dependencies among a group of libraries always form a directed acyclic graph (DAG).

The R6RS top-level program below uses `fact` from library `(A)` to print the factorial of 5.

```
(import (rnrs) (A))
(write (fact 5))
```

All top-level programs begin with an `import` form that lists the libraries upon which it relies. As with a `library` body, the only identifiers visible within a top-level program's body are those imported into the program or defined within the program. A top-level-program body is identical to a library body.[6]

[5]This description suppresses several details of the syntax, such as support for library versioning, renaming of imports or exports, identifiers exported indirectly via the expansion of a macro, and the ability to export other kinds of identifiers, such as record names.

[6]Definitions and initialization expressions can be interleaved in a top-level-program body, but this is a cosmetic difference of no importance to our discussion.

The definitions and initialization expressions within the body of a library or top-level program are evaluated in sequence. The definitions can, however, be mutually recursive. The resulting semantics can be expressed as a `letrec*`, which is a variant of `letrec` that evaluates its right-hand-side expressions in order.

4.2.1.1. Library phasing. Together, Figures 4.1, 4.2, and 4.3 illustrate how the use of macros can lead to the need for phasing between libraries. The `(tree)` library implements a basic set of procedures for creating, identifying, and modifying simple tree structures built using a tagged vector. Each tree node has a value and list of children, and the library provides accessors for obtaining the value of the node and the children. As with library `(A)`, `(tree)` exports only run-time (variable) bindings.

Library `(tree constants)` defines a macro that can be used to create constant (quoted) tree structures and three variables bound to constant tree structures. The `quote-tree` macro does not simply expand into a set of calls to `make-tree` because that would create (non-constant) trees at run time. Instead, it directly calls `make-tree` *at expansion time* to create constant tree structures. This sets up a compile-time dependency for `(tree constants)` on the run-time bindings of `(tree)`.

Finally, the top-level program shown in Figure 4.3 uses the exports of both the `(tree)` library and the `(tree constants)` library. Because it uses `quote-tree`, it depends upon the run-time exports of both libraries at compile time and at run time.

```
(library (tree)
(export make-tree tree? tree-value
          tree-children)
  (import (rnrs))
  (define tree-id #xbacca)
    (define make-tree
      (case-lambda
    [() (make-tree #f '())]
    [(v) (make-tree v '())]
  [(v c) (vector tree-id v c)]))
      (define tree?
        (lambda (t)
        (and (vector? t)
  (eqv? (vector-ref t 0) tree-id))))
    (define tree-value
(lambda (t) (vector-ref t 1)))
  (define tree-children
(lambda (t) (vector-ref t 2))))
```

FIGURE 4.1. The `(tree)` library, which implements a tree data structure.

```
(library (tree constants)
(export quote-tree t0 t1 t2)
   (import (rnrs) (tree))
 (define-syntax quote-tree
         (lambda (x)
        (define q-tree-c
           (lambda (x)
          (syntax-case x ()
              [(v c* ...)
             (make-tree #'v
           (map q-tree-c #'(c* ...)))]
            [v (make-tree #'v)])))
          (syntax-case x ()
         [(_) #'(quote-tree #f)]
         [(quote-tree v c* ...)
           #`'#,(make-tree #'v
          (map q-tree-c #'(c* ...)))])))
  (define t0 (quote-tree))
 (define t1 (quote-tree 0))
(define t2 (quote-tree 1 (2 3 4) (5 6 7))))
```

FIGURE 4.2. The `(tree constants)` library, which defines a mechanism for creating constant trees and a few constant trees of its own.

```
(import (rnrs) (tree) (tree constants))
          (define tree->list
               (lambda (t)
            (cons (tree-value t)
       (map tree->list (tree-children t)))))
          (write (tree->list t0))
          (write (tree->list t1))
(write (tree-value (car (tree-children t2))))
  (write (tree->list (quote-tree 5 (7 9))))
```

FIGURE 4.3. A program that uses the `(tree)` and `(tree constants)` libraries.

The possibility that one library's compile-time or run-time exports might be needed to compile another library sets up a *library phasing* problem that must be solved by the implementation. We say that a library's compile-time exports (i.e., macro definitions) comprise its *visit code*, and its run-time exports (i.e., variable definitions and initialization expressions) comprise its *invoke code*. When a library's compile-time exports are needed (to compile another library or top-level program), we say that the library must be *visited*, and when a library's run-time exports are needed (to compile or run another library or top-level program), we say that the library must be *invoked*.

In the tree example, the library `(tree)` is invoked when the library `(tree constants)` is compiled because the `quote-tree` forms in `(tree constants)` cannot be expanded without the run-time

exports of `(tree)`. For the same reason, library `(tree)` is invoked when the top-level program in Figure 4.3 is compiled. Library `(tree constants)` is visited when the top-level program is compiled due to the use of `quote-tree`. Finally, both libraries are invoked when the top-level program is run because the run-time bindings of both are used.

The tree example takes advantage of implicit phasing [**55**]. R6RS also allows an implementation to require explicit phase declarations as part of the `import` syntax. The `library-group` form described in this chapter and its implementation are not tied to either phasing model. As such, this chapter contains no further discussion of the differences between implicit and explicit phasing.

4.2.2. Library implementation. The compiled form of a library consists of metadata, compiled visit code, and compiled invoke code. The metadata represents information about the library's dependencies and exports, among other things. The compiled visit code evaluates the library's macro-transformer expressions and sets up the bindings from keywords to transformers. The compiled invoke code evaluates the right-hand sides of the library's variable definitions, sets up the bindings from variables to their values, and evaluates the initialization expressions.

When the first import of a library is seen, a *library manager* locates the library, loads it, and records its metadata, visit code, and invoke code in a *library record* data structure, as illustrated for libraries `(tree)` and `(tree constants)` in Figure 4.4. The metadata consist of the library's name, a unique identifier (UID), a list of exported identifiers, a list of libraries that must be invoked before the library is visited, and a list of libraries that must be invoked before the library is invoked. The UID uniquely identifies each compilation instance of a library and is used to verify that other compiled libraries and top-level programs are built against the same compilation instance. In general, when a library or top-level program is compiled, it must be linked only with the same compilation instance of an imported library. An example that illustrates why this is necessary is provided in Section 4.3.3.

Subsequent imports of the same library do not cause the library to be reloaded, although in our implementation, a library can be reloaded explicitly during interactive program development.

Once a library has been loaded, the expander uses the library's metadata to determine the library's exports. When a reference to an export is seen, the expander uses the metadata to determine whether it is a compile-time export (keyword) or run-time export (variable). If it is a compile-time export, the expander runs the library's visit code to establish the keyword bindings. If it is a run-time export, the expander's action depends on the "level" of the code that is being expanded. If the code is run-time code, the expander merely records that the library or program being expanded has an

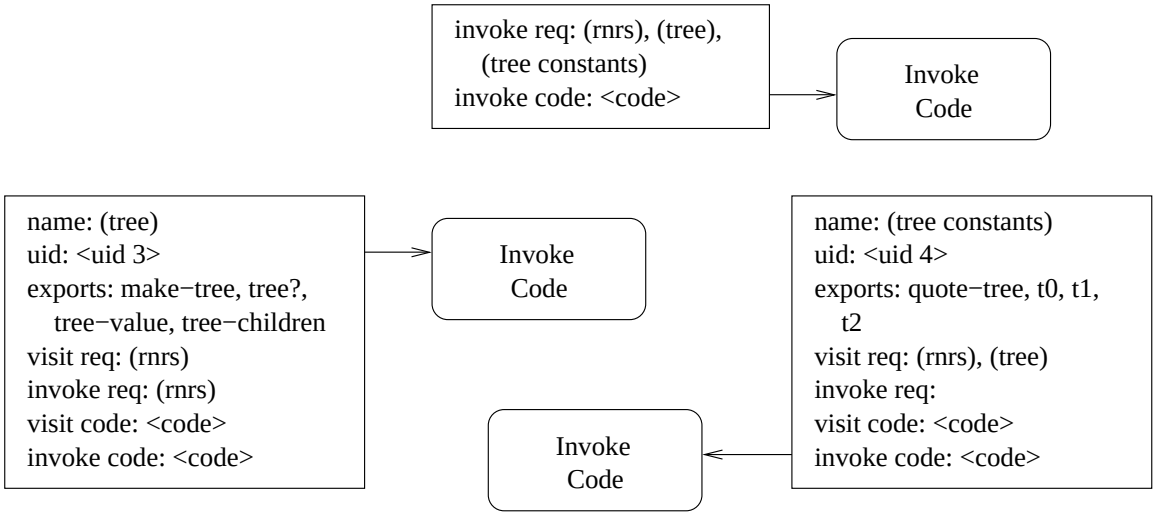

FIGURE 4.4. Library records for the `(tree)` and `(tree constants)` libraries and a program record for our program.

invoke requirement on the library. If the code is expand-time code (i.e., code within a transformer expression on the right-hand side of a `define-syntax` or other keyword binding form), the expander records that the library or program that is being expanded has a visit requirement on the library; and the expander also runs the library's invoke code to establish its variable bindings and perform its initialization.

Because programs have no exports, they do not have visit code and do not need most of the metadata associated with a library. Thus, a program's representation consists only of invoke requirements and invoke code, as illustrated at the top of Figure 4.4. In our implementation, a program record is never recorded anywhere, as the program is invoked as soon as it is loaded.

As noted in Section 4.2.1, library and top-level-program bodies are evaluated using `letrec*` semantics. Thus, the invoke code produced by the expander for a library or top-level program is structured as a `letrec*`, as illustrated below for library `(tree)`, with — used to represent the definition right-hand-side expressions, which are simply expanded versions of the corresponding source expressions.

```
(letrec* ([make-tree —]
          [tree? —]
          [tree-value —]
          [tree-children —])
  (set-top-level! $make-tree make-tree)
  (set-top-level! $tree? tree?)
  (set-top-level! $tree-value tree-value)
  (set-top-level! $tree-children tree-children))
```

If the library contained initialization expressions, they would appear just after the `letrec*` bindings. If the library contained unexported variable bindings, they would appear in the `letrec*` along with the exported bindings.

We refer to the identifiers `$make-tree`, `$tree?`, `$tree-value`, and `$tree-children` as *library globals*. These are the handles by which other libraries and top-level programs are able to access the exports of a library. In our system, library globals are implemented as ordinary top-level bindings in the sense of the Revised^5 Report on Scheme (R5RS) [**64**]. To avoid name clashes with other top-level bindings or with other compilation instances of the library, library globals are generated symbols (gensyms). In fact, the list of exports is not as simple as portrayed in Figure 4.4, as the list of exports must identify the externally visible name, e.g., `make-tree`, whether the identifier is a variable or keyword and, for variables, the generated name, e.g., the gensym represented by `$make-tree`.

It is possible to avoid binding the local names, e.g., `make-tree`, and, instead, directly set only the global names, e.g., `$make-tree`. Binding local as well as global names enables the compiler to perform the optimizations described in Section 4.1 that involve references to the library's exported variables within the library itself. Our compiler is not able to perform such optimizations when they involve references to top-level variables, as it is generally impossible to prove that a top-level variable's value never changes, even with whole-program analysis, due to the potential use of `eval`. We could introduce a new class of immutable variables to use as library globals, but this would cause problems in our system if a compiled library is ever explicitly reloaded. It is also easier to provide the compiler with code it already knows how to optimize than to teach it how to deal with a new class of immutable top-level variables.

4.3. The `library-group` Form

With this basic understanding of how libraries work and are implemented, we are ready to look at the `library-group` form. This section provides a description of the form, its usage, what the expander should produce for the form, and how the expander does so, as well as a description of a more portable variation of the expansion.

4.3.1. Usage. Both the `(tree)` and `(tree constants)` libraries are required when the top-level program that uses them is run. If the program is an application to be distributed, the libraries would have to be distributed along with the program. Because the libraries and program are compiled separately, there is no opportunity for the compiler to optimize across the boundaries and no chance

for the compiler to detect ahead of time whether one of the procedures exported by `(tree)` is used improperly by the program. The `library-group` form is designed to address all of these issues.

Syntactically, a `library-group` form is a wrapper for a set of `library` forms and, optionally, a top-level program. Here is how it might look for our simple application, with — used to indicate portions of the code that have been omitted for brevity.

```
(library-group
  (library (tree) —)
  (library (tree constants) —)
  (import (rnrs) (tree) (tree constants))
  (define tree->list
    (lambda (t)
      (cons (tree-value t)
            (map tree->list (tree-children t)))))
  (write (tree->list t0))
  (write (tree->list t1))
  (write (tree-value (car (tree-children t2))))
  (write (tree->list (quote-tree 5 (7 9)))))
```

The following grammar describes the `library-group` syntax:

library-group	⟶	`(library-group` *lglib*`*` *lgprog*`)`
	|	`(library-group` *lglib*`*)`
lglib	⟶	*library* | `(include` *filename*`)`
lgprog	⟶	*program* | `(include` *filename*`)`

where *library* is an ordinary R6RS `library` form and *program* is an ordinary R6RS top-level program. A minor but important twist is that a library or the top-level program, if any, can be replaced by an `include` form that names a file that contains that library or program.[7] In fact, we anticipate that this will be done more often than not so the existing structure of a program and the libraries it uses are not disturbed. In particular, when `include` is used, the existence of the `library-group` form does not interfere with the normal library development process or defeat the purpose of using libraries to organize code into separate logical units. So, our simple application might instead look like:

```
(library-group
  (include "tree.sls")
  (include "tree/constants.sls")
  (include "app.sps"))
```

[7]An included file can contain multiple libraries or even one or more libraries and a program, but we anticipate that each included file typically contains just one library or program.

In the general case, a `library-group` packages together a program and multiple libraries. There are several interesting special cases. In the simplest case, the `library-group` form can be empty, with no libraries and no program specified, in which case it is compiled into nothing. A `library-group` form can also consist of just the optional top-level program form. In this case, it is simply a wrapper for the top-level program that it contains, as `library` is a wrapper for libraries. Similarly, the `library-group` form can consist of a single `library` form, in which case it is equivalent to just the `library` form by itself. Finally, we can have just a list of `library` forms, in which case the `library-group` form packages together libraries only, with no program code.

A `library-group` form is not required to encapsulate all of the libraries upon which members of the group depend. For example, we could package together just `(tree constants)` and the top-level program:

```
(library-group
  (include "tree/constants.sls")
  (include "app.sps"))
```

leaving `(tree)` as a separate dependency of the library group. This is important, as the source for some libraries might be unavailable. In this case, a library group contains just those libraries for which source is available. The final distribution can include any separate binary libraries. Conversely, a `library-group` form can contain libraries upon which neither the top-level program (if present) nor any of the other libraries explicitly depend, e.g.:

```
(library-group
  (include "tree.sls")
  (include "tree/constants.sls")
  (include "foo.sls")
  (include "app.sps"))
```

Even for whole programs packaged in this way, including an additional library might be useful if the program might use `eval` to access the bindings of the library at run time. This supports the common technique of building modules that might or might not be needed into an operating system kernel, web server, or other program. The advantage of doing so is that the additional libraries become part of a single package, and they benefit from cross-library error checking and optimization for the parts of the other libraries that they use. The downside is that libraries included but never used might still have their invoke code executed, depending on which libraries in the group are invoked. This is the result of combining the invoke code of all the libraries in the group. The programmer has the responsibility and opportunity to decide which libraries are profitable to include.

Apart from the syntactic requirement that the top-level program, if present, must follow the libraries, the `library-group` form also requires that each library be preceded by any other library in the group that it imports. So, for example:

```
(library-group
  (include "tree/constants.sls")
  (include "tree.sls")
  (include "app.sps"))
```

would be invalid, because `(tree constants)` imports `(tree)`. One or more appropriate orderings are guaranteed to exist because R6RS libraries are not permitted to have cyclic import dependencies.

The expander could determine an ordering based on the `import` forms (including local `import` forms) that it discovers while expanding the code. We give the programmer complete control over the ordering, however, so that the programmer can resolve dynamic dependencies that arise from invoke-time calls to `eval`. Another solution would be to reorder only if necessary, but we have so far chosen not to reorder so as to maintain complete predictability.

Libraries contained within a `library-group` form behave like their stand-alone equivalents, except that the invoke code of the libraries is fused.[8] Fusing the code of the enclosed libraries and top-level program facilitates compile-time error checking and optimization across the library and program boundaries. If compiled to a file, the form also produces a single object file. In essence, the `library-group` form changes the basic unit of compilation from the library or top-level program to the `library-group` form, without disturbing the enclosed (or included) libraries or top-level programs.

A consequence of fusing the invoke code is that, the first time that a library in the group is invoked, the libraries up to and including that library are invoked as well, along with any side effects such invoking might entail. In cases where all of the libraries in the group would be invoked anyway, such as when a top-level program that uses all of the libraries is run, this situation is no different from the stand-alone behavior.

Fusing the invoke code creates a more subtle difference between grouped and stand-alone libraries. The import dependencies of a group of R6RS libraries must form a DAG, i.e., must not involve cycles. An exception is raised at compile time for static cyclic dependencies and at run time for dynamic cyclic dependencies that arise via `eval`. When multiple libraries are grouped together, a

[8]Visit code is not fused as there is no advantage in doing so.

synthetic cycle can arise, just as cycles can arise when arbitrary nodes in any DAG are combined. We address the issue of handling dynamic cycles in more depth in the next subsection.

4.3.2. Anticipated expander output. This section contains a description of what we would like the expander to produce for the `library-group` form and how the expander deals with import relationships that require one library's run-time exports to be available for the expansion of another library within the group.

As noted in Section 4.2, the explicit import dependencies among libraries must form a DAG and, as shown in Section 4.2.2, the invoke code of each library expands independently into a `letrec*` expression. This leads to an expansion of `library-group` forms as nested `letrec*` forms, where each library expands to a `letrec*` form that contains the libraries that follow it in the group. The code for the top-level program is nested inside the innermost `letrec*` form. Libraries are nested in the order provided by the programmer in the `library-group` form.

Figure 4.5 shows the result of this nesting of `letrec*` forms for the first library group defined in Section 4.3.1. This is a good first cut. The references to each library global properly follow the assignment to it, which remains properly nested within the binding for the corresponding local variable. Unfortunately, this form does not allow the compiler to analyze and optimize across library boundaries, as the inner parts of the nested `letrec*` refer to the global rather than to the local variables.

To address this shortcoming, the code must be rewired to refer to the local variables instead, as shown in Figure 4.6. With this change, the invoke code of the library group now forms a single compilation unit for which cross-library error checking and optimization are possible.

Another issue remains. Loading a library group should not automatically execute the shared invoke code. To address this issue, the code is abstracted into a separate procedure, p, called from the invoke code stored in each of the library records. Rather than running the embedded top-level-program code, p returns a thunk that can be used to run that code. This thunk is ignored by the library invoke code, but it is used to run the top-level program when the library group is used as a top-level program. The procedure p for the tree library group is shown in Figure 4.7.

Unfortunately, this expansion can lead to synthetic cycles in the dependency graph of the libraries. Figure 4.8 shows three libraries with simple dependencies: `(C)` depends on `(B)`, which in turn depends on `(A)`.

```
(letrec* ([tree-id —]
          [make-tree —]
            [tree? —]
          [tree-value —]
        [tree-children —])
  (set-top-level! $make-tree make-tree)
      (set-top-level! $tree? tree?)
  (set-top-level! $tree-value tree-value)
(set-top-level! $tree-children tree-children)
       (letrec* ([t0 —]
                 [t1 —]
                [t2 —])
     (set-top-level! $t0 t0)
     (set-top-level! $t1 t1)
     (set-top-level! $t2 t2)
      (letrec* ([tree->list
                 (lambda (t)
            (cons ($tree-value t)
                (map tree->list
             ($tree-children t))))])
      (write (tree->list $t0))
      (write (tree->list $t1))
         (write (tree-value
         (car (tree-children $t2))))
(write (tree->list (quote tree constant))))))
```

FIGURE 4.5. A nested `letrec*` for the library group that includes the `(tree)` library, the `(tree constants)` library, and our program, with — indicating code that has been omitted for brevity.

We could require the programmer to include library `(B)` in the library group, but a more general solution that does not require this is preferred. The central problem is that `(B)` needs to be run after the invoke code for library `(A)` is finished and before the invoke code for library `(C)` has started. This can be solved by marking library `(A)` as invoked once its invoke code is complete and explicitly invoking `(B)` before `(C)`'s invoke code begins. Figure 4.10 shows what this invoke code might look like.

This succeeds when `(A)` or `(C)` are invoked first but results in a cycle when `(B)` is invoked first. Effectively, the library group invoke code should stop once `(A)`'s invoke code has been executed. Wrapping each library in a `lambda` that takes the UID of the library being invoked accomplishes this. When a library group is invoked, the UID informs the invoke code where to stop and returns any nested library's surrounding `lambda` as the restart point. Figure 4.11 shows this corrected expansion of the library group that contains `(A)` and `(C)`. The invoke code for an included program would replace the innermost `nested-lib` and be called when `#f` is passed in place of the UID.

```
(letrec* ([tree-id —]
          [make-tree —]
            [tree? —]
          [tree-value —]
        [tree-children —])
  (set-top-level! $make-tree make-tree)
      (set-top-level! $tree? tree?)
  (set-top-level! $tree-value tree-value)
(set-top-level! $tree-children tree-children)
       (letrec* ([t0 —]
                 [t1 —]
                [t2 —])
     (set-top-level! $t0 t0)
     (set-top-level! $t1 t1)
     (set-top-level! $t2 t2)
      (letrec* ([tree->list
                 (lambda (t)
                (cons (tree-value t)
                    (map tree->list
                 (tree-children t))))])
      (write (tree->list t0))
      (write (tree->list t1))
        (write (tree-value
         (car (tree-children t2))))
(write (tree->list (quote tree constant))))))
```

FIGURE 4.6. A nested `letrec*` for our library group that includes the `(tree)` library, the `(tree constants)` library, and our program, with `library-global` references replaced by local-variable references.

In addition to addressing the issues in the invoke code, we would also like to ensure that libraries in the group are properly installed in the library manager. For the most part, libraries in the group can be handled like stand-alone libraries. Metadata and visit code are installed in the library manager as normal. The invoke code is the only twist. We would like to ensure that each library in the library group is invoked only once, the first time it or one of the libraries below it in the group is invoked. Thus, each library is installed with the shared invoke procedure described above. Figure 4.12 shows how our library records are updated from Figure 4.4 to support the shared invoke code. Figure 4.13 shows this final expansion for our tree library group. If the optional program were not supplied, the call to the p thunk at the bottom would be omitted. When the optional program is supplied, it always executes when the library group is loaded. Programmers who wish to use the library group separately can create two versions of the library group, one with the top-level program and one without.

```
                (lambda ()
              (letrec* ([tree-id —]
                        [make-tree —]
                          [tree? —]
                        [tree-value —]
                       [tree-children —])
           (set-top-level! $make-tree make-tree)
               (set-top-level! $tree? tree?)
          (set-top-level! $tree-value tree-value)
       (set-top-level! $tree-children tree-children)
                    (letrec* ([t0 —]
                              [t1 —]
                             [t2 —])
                   (set-top-level! $t0 t0)
                   (set-top-level! $t1 t1)
                   (set-top-level! $t2 t2)
                          (lambda ()
                    (letrec* ([tree->list
                               (lambda (t)
                              (cons (tree-value t)
                                 (map tree->list
                                (tree-children t))))])
                    (write (tree->list t0))
                    (write (tree->list t1))
                      (write (tree-value
                       (car (tree-children t2))))
                      (write (tree->list
                      (quote tree constant)))))))))
```

FIGURE 4.7. The final invoke code expansion target for the library group that includes the `(tree)` library, the `(tree constants)` library, and our program.

```
(library (A)
  (export x)
  (import (rnrs))
  (define x 5))
(library (B)
  (export y)
  (import (rnrs) (A))
  (define y (+ x 5)))

(library (C)
  (export z)
  (import (rnrs) (B))
  (define z (+ y 5)))
```

FIGURE 4.8. Three simple libraries, `(A)`, `(B)`, and `(C)`, with simple dependencies.

```
(library-group (library (A) —) (library (C) —))
```

FIGURE 4.9. A `library-group` form containing the `(A)` and `(C)` libraries.

```
(lambda ()
 (letrec* ([x 5])
(set-top-level! $x x)
 ($mark-invoked! 'A)
($invoke-library '(B) '() 'B)
   (letrec* ([z (+ y 5)])
    (set-top-level! $z z)
    ($mark-invoked! 'C))))
```

FIGURE 4.10. Expansion of library group marking `(A)` as invoked and invoking `(B)`.

```
(lambda (uid)
 (letrec* ([x 5])
(set-top-level! $x x)
 ($mark-invoked! 'A)
  (let ([nested-lib
       (lambda (uid)
($invoke-library '(B) '() 'B)
   (letrec* ([z (+ y 5)])
     (set-top-level! $z z)
      ($mark-invoked! 'C)
  (let ([nested-lib values])
       (if (eq? uid 'C)
            nested-lib
           (nested-lib uid)))))])
   (if (eq? uid 'A)
        nested-lib
       (nested-lib uid)))))
```

FIGURE 4.11. Final expansion for correct library group with the `(A)` and `(C)` libraries.

4.3.3. Implementation. A major challenge in producing the residual code shown in the preceding section is that the run-time bindings for one library might be needed while compiling the code for another library in the group. A potential simple solution to this problem is to compile and load each library before compiling the next one in the group. This causes the library (and any similar library) to be compiled twice, but that is not a serious concern if the compiler is fast or if the `library-group` form is used only in the final stage of an application's development to prepare the final production version.

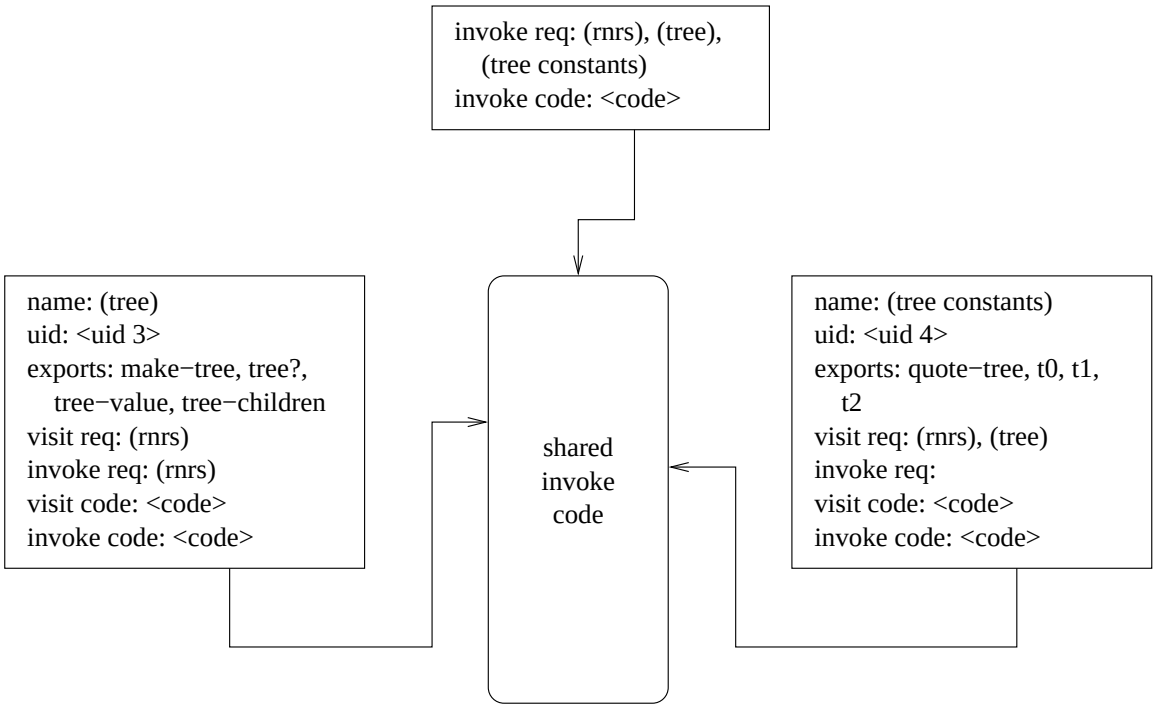

FIGURE 4.12. Library and program records for the library group, showing the shared invoke code run when either the `(tree)` or `(tree constant)` library is invoked or when the top-level program is run.

Unfortunately, this simple solution does not work because the first compilation of the library may be fatally incompatible with the second. This can arise for many reasons, all having to do ultimately with two facts. First, macros can change much of the nature of a library, including the internal representations used for its data structures and even whether an export is defined as a keyword or as a variable. Second, because macros can take advantage of the full power of the language, the transformations that they perform can be affected by the same things that affect run-time code, including, for example, information in a configuration file, state stored elsewhere in the file system by earlier uses of the macro, or even a random number generator.

For example, via a macro that flips a coin, e.g., checks to see whether a random-number generator produces an even or odd answer, the `(tree)` library might in one case represent trees as tagged lists and in another as tagged vectors. If this occurs, the constant trees defined in the `(tree constants)` library and in the top-level program would be incompatible with the accessors used at run time. While this is a contrived and whimsical example, such things can happen, and we are obligated to handle them properly to maintain consistent semantics between separately compiled libraries and libraries compiled as part of a library group.

```
(let ([p (let ([proc (lambda (uid)
                       (letrec* ([tree-id —]
                                 [make-tree —]
                                   [tree? —]
                                 [tree-value —]
                               [tree-children —])
                         (set-top-level! $make-tree make-tree)
                                        —
                                ($mark-invoked! 'tree)
                           (let ([nested-lib (lambda (uid)
                                               (letrec* ([t0 —]
                                                         [t1 —]
                                                        [t2 —])
                                                 (set-top-level! $t0 t0)
                                                        —
                                                ($mark-invoked! 'constants)
                                               (let ([nested-lib (lambda (uid)
                                                                   ($invoke-library
                                                                    '(tree constants)
                                                                     '() 'constants)
                                                                   ($invoke-library
                                                                   '(tree) '() 'tree)
                                                                       (letrec*
                                                                    ([tree->list —])
                                                                         —))])
                                                  (if (eq? uid 'constants)
                                                          nested-lib
                                                    (nested-lib uid))))])
                                 (if (eq? uid 'tree)
                                       nested-lib
                                  (nested-lib uid))))])
                    (lambda (uid) (set! proc (proc uid))))])
       ($install-library '(tree) '() 'tree
       '(#[libreq (rnrs) (6) $rnrs]) '() '()
           void (lambda () (p 'tree)))
($install-library '(tree constants) '() 'constants
             '(#[libreq (tree) () tree]
               #[libreq (rnrs) (6) $rnrs])
          '(#[libreq (tree) () tree]) '()
                    (lambda ()
           (set-top-level! $quote-tree —))
             (lambda () (p 'constants)))
                     (p #f))
```

FIGURE 4.13. Final expansion of the library group containing the `(tree)` library, the `(tree constants)` library, and our program.

On the other hand, we cannot entirely avoid compiling the code for a library whose run-time exports are needed to compile another part of the group if we are to produce the run-time code that we hope to produce. The solution is for the expander to *expand* the code for each library only once, as it is

seen, as though the library were compiled separately from all of the other libraries. If the library must be invoked to compile another of the libraries or the top-level program, the expander runs the invoke code through the rest of the compiler and evaluates the result. Once all of the libraries and the top-level program have been expanded, the expander can merge and rewrite the expanded code for all of the libraries to produce the code described in the preceding section, and then allow the resulting code to be run through the rest of the compiler. Although some of the libraries might be put through the rest of the compiler more than once, each is expanded exactly once. Assuming that the rest of the compiler is deterministic, this prevents the types of problems that arise if a library is expanded more than once.

To perform this rewiring, the library must be abstracted slightly so that a mapping from the exported identifiers to the lexical variables can be maintained. With this information, the code can be rewired to produce the code in Figure 4.13.

Because a library's invoke code might be needed to expand another library in the group, libraries in the group are installed as stand-alone libraries during expansion and are then replaced by the library group for run time. This means that the invoke code for a library might be run twice in the same Scheme session, once during expansion and once during execution. Multiple invocations of a library are permitted by the R6RS. Indeed, some implementations always invoke a library one or more times at compile time and again at run time to prevent state setup at compile time from being used at run time.

This implementation requires the expander to walk through expanded code and to convert library-global references into lexical-variable references. Expanded code is typically in some compiler-dependent form that the expander would not normally need to traverse, and we might want a more portable solution to this problem. One alternative to the code walk is to wrap the expanded `library` in a `lambda` expression, with formal parameters for each library global referenced within the library.

4.4. Automatic Cross-Library Optimization

The `library-group` form provides optimizations akin to whole-program optimization, bringing together more source code upon which the compiler can operate. Even when the `library-group` form is not used, however, we would like to provide some of the benefits of cross-library optimization. Naively, one approach to cross-library optimization would be to include a representation of the source code for the library so that this source code can be incorporated when the library is imported and would work similarly to that of libraries combined with the `library-group` form. The challenge in

this approach is that any shared-mutable state setup by the library cannot be duplicated without changing the semantics that results from a library being imported into the same session through two different libraries. The other downside of this approach is that potentially large pieces of source code that would not normally be copied by the source optimizer, which is set up to limit code growth due to inlining, would be stored but effectively never used, needlessly increasing the size of library binaries.

Instead, our approach attaches a constant or a representation of the source code for a procedure to an identifier only when the constant is *copyable* or the procedure *inlinable*. A constant is considered *copyable* when copying it will not change the semantics of a program that uses the constant. The driving decider for this is how `eq?` handles the constant. A structured constant, such as a vector or a pair, cannot be copied because the constant must be `eq?` to itself, and copying the constant would break this property. A procedure is considered *inlinable* when it contains no free variables, only copyable constants, and fits within the *score limit* when the library is compiled. The *score limit* controls the amount of copying that the source inliner will perform when inlining occurs. It is normally based on the size of the code after inlining but must be based on the size before inlining here, as we are operating without knowledge of the call site. Limiting inlinable procedures to those without free variables both avoids potential problems with referencing an identifier that is not bound in the context where the source is copied and any copying of shared mutable state. At this point in the compiler, variables are specifically local variables, so references to primitives or globals are not considered free. The one exception to this is references to imported library identifiers. Allowing references to library identifiers would require that library dependencies be updated in libraries that import the one being compiled. There is currently no facility for doing this.

4.4.1. Implementation. The implementation of automatic cross-library optimization requires that both the expander and the source optimizer be aware of the optimization. As described in Section 4.2.2, definitions in a library are bound by a `letrec*` form, and exported identifiers are then set in the top-level environment to the value of the corresponding local variable. To enable automatic cross-library optimization, the internal representation of a library global is extended to contain a mutable field that stores the optimization information for the identifier. During expansion of the library, this field is set to `#f`, which indicates that no optimization information is available. The expander also generates a node in the internal representation of the output of the library to indicate to the source optimizer that the contained expression is related to the given library identifier. This

breadcrumb is the only piece of information left about the library after the expander is finished expanding the library.

When the source optimizer encounters a cross-library optimization node, it first performs source optimization on the expression contained in the node and then inspects the result. If the result is a copyable constant or an inlinable procedure, the optimization field of the associated library export is modified to contain the internal (record) representation of the expression. This sets up the cross-library optimization, as the internal representation of a library global is written to the binary output file for use when the library is imported.

When a library is imported, the expander is responsible for completing the cross-library optimization by replacing references to library globals with the optimization information, when it is available. Performing the replacement in the expander is convenient and allows library dependencies to be more tightly computed, as there is no dependence on the library if all of the library identifiers used from the library are optimized away. When a library global is not in call position in the library source but is moved into call or test position through inlining, the source optimizer replaces the library global with the optimization information. This allows inlining that would not be possible using only the expander but does not allow library dependencies, which are fixed by the time source optimization occurs, to be dropped.

The implicit cross-library optimization is inherently limited. Constructed constants cannot be copied because, if any two libraries with the same constructed constant were imported, the constants would no longer be `eq?` to each other. Only procedure expressions exported from the library that contain no free variables and no references to library globals are eligible for external inlining, as they will be lifted out of their lexical context and do not carry the set of dependencies needed for the library globals. These procedures are further limited by the size of the expression, as the inlining process would normally count only the size of the procedure after inlining had occurred. This might be allowed to be much larger if the argument expressions to the procedure are large.

4.5. Empirical Evaluation

One of the goals of the `library-group` form is to enable cross-library optimizations to occur. Optimizations such as procedure inlining are known to result in significant performance benefits [**90**]. By using the `library-group` form, a programmer enables a compiler that supports these optimizations to apply them across library boundaries. This section presents a characterization of the types of

programs that we expect to show performance benefits. Even when there are no performance benefits, programs still benefit from the single binary output file and cross-library compile-time error checking.

Even when the `library-group` form is not used, the automatic cross-library optimizations enable some procedure inlining and constant folding. This does not have the benefit of creating a single binary; however, the optimization occurs without any intervention from the programmer. Used together, optimizations can occur across library group boundaries as well.

In general, programs and libraries with many cross-library procedure calls are expected to benefit the most. As an example, imagine a compiler where each pass is called only once and is defined in its own library. Combining these libraries into a library group is unlikely to yield performance benefits, as the number of cross-library procedure calls is relatively small. However, if the passes of this compiler use a common record structure to represent code and a library of helpers for decomposing and reconstructing these records, combining the compiler pass libraries and the helper library into a single library group can benefit compiler performance significantly.

To illustrate when performance gains are expected, we present two example libraries, both written by Eduardo Cavazos and tested in Chez Scheme Version 8.9.5 [**33**]. We present only two tests here, because while the `library` form has been successful for providing libraries such as the Scheme SRFIs, not many full applications or benchmarks have been developed that make use of the library form. The first program [**24**] implements a set of tests for the "Mathematical Pseudo Language" [**27**, **28**] (MPL), a symbolic math library. The second uses a library for indexable sequences [**23**] to implement a matrix multiply algorithm [**34**].

Many small libraries comprise the MPL library. Each basic mathematical function, such as `+`, `/`, and `cos`, uses pattern matching to decompose the mathematical expression passed to it to select an appropriate simplification, if one exists. The pattern matcher, provided by another library [**42**], avoids cross-library calls, as it is implemented entirely as a macro. The mathematical libraries, however, make use of operations defined as procedures in other libraries, which cannot be inlined across library boundaries without one of the optimizations described in this chapter. The test library also makes many cross-library calls to both the math libraries and the testing support library. Thus, there are many cross-library calls that can be eliminated by using the `library-group` form. Using the `library-group` form results in a 24% speed-up over the separately compiled version, with automatic cross-library optimization disabled. Because many of the mathematical operations are also inlinable procedures, the automatic cross-library optimization also optimizes the MPL tests

well. When the automatic cross-library optimizations are enabled, it results in a 24% speed-up over the version without the automatic cross-library optimization. Using the cross-library optimization and the `library-group` form together does not result in further improvements.

The matrix-multiply example uses a `vector-for-each` form that provides the loop index to its procedure argument, from the `indexable-sequence` library. The library abstracts standard data structure iteration functions that provide constructors, accessors, and a length function. The operations it creates, however, are implemented as macros, so operations defined by these libraries are expanded into basic operations at their use sites. The `matrix-multiply` function makes three nested calls to `vector-for-each-with-index`, which expand in the matrix multiple library into inline loops. A test program calls matrix-multiply on 50 x 50, 100 x 100, and 500 x 500 matrices. The calls to the multiply operation are cross-library calls, but the majority of the work of performing matrix multiple occurs entirely in the matrix multiple library, so we do not expect much benefit from cross-library optimizations. Thus, compiling the program with the `library-group` form showed only a negligible performance gain over the version with automatic cross-library optimization disabled. Enabling the automatic cross-library optimization also does not improve the performance.

In both of our example programs, the difference in time between compiling the program as a set of individual libraries with implicit cross-library optimization enabled and as a single library group is negligible. Despite this, it is possible to construct a contrived example where the implicit cross-library optimization is not applicable, but the `library-group` form is.

4.6. Related Work

Packaging code into a single distributable is not a new challenge, and previous dialects of Scheme needed a way to provide a single binary for distribution. Our system, PLT Racket, and others provide mechanisms for packaging up and distributing collections of compiled libraries and programs. These are packaging facilities only and do not provide the cross-library optimization or compile-time error checking provided by the `library-group` form.

Ikarus [**51**] uses Waddell's source optimizer [**89**, **90**] to perform some of the same interprocedural optimizations as performed in our system. In both systems, these optimizations previously occurred only within a single compilation unit, e.g., a top-level expression or library. The `library-group` form allows both to perform cross-library and even whole-program optimization. The Stalin [**82**] Scheme compiler supports aggressive whole-program optimization when the whole program is presented to it, but it does not support R6RS libraries or anything similar. If at some point it does support

R6RS libraries, the `library-group` form would be a useful addition. MIT Scheme [**58**] allows the programmer to mark a procedure inlinable, and inlining of procedures so marked occurs across file boundaries. MIT Scheme does not support R6RS libraries, and inlining, while important, is only one of many optimizations enabled when the whole program is made available to the compiler. Thus, as with Stalin, if support for R6RS libraries were added to MIT Scheme, the `library-group` form would be a useful addition.

Although the `library-group` mechanism is orthogonal to the issue of explicit versus implicit phasing, the technique we use to make a library's run-time bindings available both independently at compile time and as part of the combined library-group code is similar to techniques Flatt uses to support separation of phases [**45**].

Outside the Scheme community, several other languages, such as Dylan, ML, Haskell, and C++, make use of library or module systems and provide some form of compile-time abstraction facility. Dylan is the closest to Scheme and is latently typed with a rewrite-based macro system [**77**]. Dylan provides both libraries and modules, where libraries are the basic compilation unit, and modules are used to control scope. The Dylan community also recognizes the benefits of cross-library inlining, and a set of common extensions allows programmers to specify when and how functions should be inlined. By default, the compiler performs intra-library inlining, but `may-inline` and `inline` specify that the compiler may try to perform inter-library inlining or that a function should always be inlined, even across library boundaries.

The Dylan standard does not include procedural macros, so run-time code from a Dylan library does not need to be made available at compile time. Nevertheless, such a facility is planned [**44**] and at least one implementation exists [**13**]. When this feature is added to existing Dylan implementations, an approach similar to that taken by the `library-group` might be needed to enable cross-library optimization.

ML functors provide a system for parameterizing modules across different type signatures, where the types needed at compile time are analogous to Scheme macros. The MLton compiler [**95**] performs whole-program compilation for ML programs and uses compile-time type information to specialize code in a functor. Because this type information is not dependent on the run-time code of other modules, it does not require a module's run-time code to be available at compile time. If the type system were extended to support dependent types, however, some of the same techniques used in the `library-group` form could be needed. Additionally, MetaML [**85**] adds staging to ML, similar to the phasing in Scheme macros. Because MetaML does not allow run-time procedures to be called in

its templates, however, it does not have the same need to make a module's run-time code available at compile time.

The Glasgow Haskell Compiler (GHC) [**1**] provides support for cross-module inlining [**87**] as well as for compile-time meta-programming through Template Haskell [**79**]. Thus, GHC achieves some of the performance benefits of the `library-group` form in a language with similar challenges, without the use of an explicit `library-group` form. A Haskell version of the `library-group` form would still be useful for recognizing when an inlining candidate is singly referenced and for enabling other interprocedural optimizations. It likely would be simpler to implement due to the lack of state at compile time.

The template system of C++ [**3**, **4**] provides a Turing-complete, compile-time abstraction facility, similar to the procedural macros found in Scheme. The language of C++ templates is distinct from C++, and run-time C++ code cannot be used during template expansion. If the template language were extended to allow C++ templates to call arbitrary C++ code, compilation might need to be handled similarly to how the `library-group` form is handled.

Another approach to cross-library optimizations is link-time optimization of object code. Several different approaches to this technique exist and are beginning to be used in compilers such as GCC [**70**] and compiler frameworks such as LLVM [**66**]. Instead of performing procedure inlining at the source level, these optimizers take object code produced by the compiler and perform optimization when the objects are linked. The GOld [**15**] link-time optimizer applies similar techniques to optimize cross-module calls when compiling Gambit-C Scheme code into C. Our decision to combine libraries at the source level is motivated by the fact that our system and others already provide effective source optimizers that can be leveraged to perform cross-library optimization.

4.7. Future Work

The `library-group` form is designed to allow programmers the greatest possible flexibility in determining which libraries to include in a library group and the order in which they should be invoked. This level of control is not always necessary, and we envision a higher-level interface to the `library-group` form that would automatically group a program with its required libraries and automatically determine an appropriate invocation order based only on static dependencies.

The `library-group` form ensures that all exports for libraries in the library group are available outside the library group. In cases where a library is not needed outside the library group, we would like to allow their exports to be dropped so that the compiler can eliminate unused code and data.

This would help reduce program bloat in cases where a large utility library is included in a program and only a small part of it is needed. We envision an extended version of the `library-group` form that specifies a list of libraries that should not be exported. The compiler should still, at least optionally, register unexported libraries to raise an exception if they are used outside the library group.

Our current implementation of the `library-group` form can lead to libraries being invoked that are not required to be invoked, based on the ordering of libraries in the group. It is possible to invoke libraries only as they are required by using a more intricate layout of library bindings, similar to the way that `letrec` and `letrec*` are currently handled [**56**]. This expansion would separate side-effect-free expressions in a library from those with side effects, running the effectful expressions only when required. This approach would require that other parts of the compiler be made aware of the `library-group` form, as the expander does not have all the information it needs to handle this effectively.

4.8. Conclusion

The `library-group` form builds on the benefits of R6RS libraries and top-level programs, which allow a single compilation unit to be created from a group of libraries and an optional top-level program. Packaging the run-time code in a single compilation unit and wiring the code together so that each part of the library group references the exports of the others via local variables allow the compiler to perform cross-library optimization and extend compile-time error checking across library boundaries. It also allows the creation of a single output binary. The implementation is designed to deliver these benefits without requiring the compiler to do any more than it already does. In this way, it represents a non-invasive feature that can be more easily incorporated into existing Scheme compilers.

In Scheme systems where the `library-group` form does not exist or where a programmer has chosen not to use it, the automatic cross-library optimization can help reap some of the benefits of the `library-group` form. In particular, automatic cross-library optimization allows Scheme record operations to be optimized across library boundaries. This removes the downside of using records with libraries. Simple operations, such as those in the MPL library, also benefit from this form of optimization.

While this work was developed in the context of Scheme, we expect that the techniques described in this chapter will become useful as other languages adopt procedural macro systems. The PLOT

language [**69**], which shares an ALGOL-like syntax with Dylan already provides a full procedural macro system, and a similar system has been proposed for Dylan [**44**]. The techniques described in this chapter might also be useful for languages with dependent-type systems that allow types to be expressed in the full source language or template meta-programming systems that allow templates to be defined through the use of the full source language.

CHAPTER 5

Closure Optimization

5.1. Introduction

First-class procedures, i.e., indefinite extent procedural objects that retain the values of lexically scoped variables, were incorporated into the design of the Scheme programming language in 1975 and, within a few years, started appearing in functional languages such as ML. It has taken many years, but they are fast becoming commonplace, with their inclusion in contemporary languages such as JavaScript and newer versions of other languages such as C# and Perl.

First-class procedures are typically represented at run time as *closures*. A closure is a first-class object that encapsulates some representation of a procedure's code (e.g., the starting address of its machine code), along with some representation of the lexical environment. In 1983, Cardelli [**22**] introduced the notion of *flat closures*. A flat closure resembles a vector, with a code slot plus one slot for each free variable.[9] The code slot holds a code pointer, which might be the address of a block of machine code that implements the procedure or some other representation of code, such as byte code in a virtual machine. Each free-variable slot holds the value of one free variable. Because the same variable's value might be stored simultaneously in one or more closures and in the original location in a register or stack, mutable variables are not directly supported by the flat-closure model. In 1987, Dybvig [**30**] addressed this for languages, such as Scheme, with mutable variables by adding a separate assignment conversion step that converts the locations of assigned variables (but not unassigned variables) into explicit heap-allocated boxes, thereby avoiding problems with duplication of values.

Flat closures have the useful property that each free variable (or location, for assigned variables) is accessible with a single indirect. This compares favorably with any mechanism that requires traversal of a nested-environment structure. The cost of creating a flat closure is proportional to the number of free variables, which is often small. When not, the cost is more than compensated for by the lower cost of free-variable reference, in the likely case that each free variable is accessed at least

[9] In this context, free variables are those references within the body of a procedure but are not bound within the procedure.

once and possibly many times. Flat closures also hold onto no more of the environment than the procedure might require and, thus, are "safe for space" [**78**]. This is important because it allows the storage manager to reclaim storage from the values of variables that are visible in the environment but not used by the procedure.

This chapter provides a description of a set of optimizations of the flat-closure model that reduce closure-creation costs and eliminate memory operations without losing the useful features of flat closures. It also presents, in detail, an algorithm that performs the optimizations and shows that the optimizations reduce run-time closure-creation and free-variable access overhead on a set of standard benchmarks by over 50%. These optimizations never do any harm, i.e., they never add allocation overhead or memory operations relative to a naive implementation of flat closures. Thus, a programmer can count on at least the performance of the straight flat-closure model and, most likely, better. The algorithm adds a small amount of compile-time overhead during closure conversion, but because it produces less code, the overhead is more than made up for by the reduced overhead in later passes of the compiler.

A key contribution of this work is the detailed description of the optimizations and their relationships. While a few of the optimizations have been performed by our compiler since 1992, descriptions of them have never been published. Various closure optimizations have been described by others [**10**, **29**, **46**, **62**, **65**, **76**, **78**, **84**], but most of the optimizations described here have not been described previously in the literature, and many are likely novel. A second key contribution is the algorithm to implement them, which also is novel.

The remainder of this chapter is organized as follows. Section 5.2 provides a description of the optimizations, and Section 5.3 provides an algorithm that implements them. Section 5.4 presents an empirical analysis that demonstrates the effectiveness of the optimizations. Section 5.5 includes related work, and Section 5.6 presents our conclusions.

5.2. The Optimizations

The closure optimizations described in this section collectively act to eliminate some closures and reduce the sizes of others. When closures are eliminated in one section of the program, the optimizations can cascade to further optimizations that allow other closures to be eliminated or reduced in size. They also sometimes result in the selection of alternate representations that occupy fewer memory locations. In most cases, they also reduce the number of indirects required to access free variables. The remainder of this section presents each optimization in turn, grouped by direct effect:

- avoiding unnecessary closures (Section 5.2.1),
- eliminating unnecessary free variables (Section 5.2.2), and
- sharing closures (Section 5.2.3).

A single algorithm that implements all of the optimizations described in this section is provided in Section 5.3.

5.2.1. Avoiding unnecessary closures. A flat closure contains a code pointer and a set of free-variable values. Depending on the number of free variables and whether the code pointer is used, we can sometimes eliminate the closure, sometimes allocate it statically, and sometimes represent it more efficiently. We consider first the case of well-known procedures.

Case 1: Well-known procedures

A procedure is *known* at a call site if the call site provably invokes that procedure's λ-expression and only that λ-expression. A *well-known* procedure is one whose value is never used except at call sites where it is known. The code pointer of a closure for a well-known procedure need never be used because, at each point where the procedure is called, the call can jump directly to the entry point for the procedure via a direct-call label associated with the λ-expression.

Depending on the number of free variables, we can take advantage of this as follows.

Case 1a: Well-known with no free variables

If the procedure has no free variables, and its code pointer is never used, the closure itself is entirely useless and can be eliminated.

Case 1b: Well-known with one free variable x

If the procedure has one free variable, and its code pointer is never used, the only useful part of the closure is the free variable. In this case, the closure can be replaced with the free variable everywhere that it is used.

Case 1c: Well-known with two free variables x and y

If the procedure has two free variables, and its code pointer is never used, it contains only two useful pieces of information, the values of the two free variables. In this case, the closure can be replaced with a pair. In our implementation, pairs occupy just two words of memory, while a closure with two free variables occupies three words.

Case 1d: Well-known with three or more free variables x ...

If the procedure has three or more free variables, but its code pointer is never used, we can choose to represent it as a closure or as a vector. The size in both cases is the same: one word for each free variable plus one additional word. The additional word for the closure is a code pointer, while the additional word for the vector is an integer length. This choice is a virtual toss-up, although storing a small constant length is slightly cheaper than storing a full-word code pointer, especially on 64-bit machines. We choose the vector representation for this reason and because it helps us share closures, as described in Section 5.2.3.

We now turn to the case where the procedure is not well known.

Case 2: Not-well-known procedures

In this case, the procedure's value might be used at a call site where the procedure is not known. That call site must jump indirectly through the closure's code pointer, as it does not know the direct-call label or labels of the closures that it might call. In this case, the code pointer is needed, and a closure must be allocated.

We consider two subcases:

Case 2a: Not well-known with no free variables

In this case, the closure is the same each time the procedure's λ-expression is evaluated, as it contains only a static code pointer. The closure can thus be allocated statically and treated as a constant.

Case 2b: Not well-known with one or more free variables x ...

In this case, a closure must be created at run time.

5.2.2. Eliminating unnecessary free variables. On the surface, it seems that a closure needs to hold the values of all of its free variables. After all, if a variable occurs free in a procedure's λ-expression, it might be referenced, barring dead code that should have been eliminated by some earlier pass of the compiler. Several cases do arise, however, in which a free variable is not needed.

Case 1: Unreferenced free variables

Under normal circumstances, a variable cannot be free in a λ-expression if it is not referenced there (or assigned, prior to assignment conversion). This case can arise after free-variable analysis has been performed, however, by the elimination of a closure under Case 1a of Section 5.2.1. Call sites that originally passed the closure to the procedure do not do so when the closure is eliminated,

and because no other references to a well-known procedure's name appear in the code, the variable should be removed from any closures in which it appears.

Case 2: Global variables

The locations of global variables, i.e., variables whose locations are fixed for an entire program run, need not be included in a closure, as the address of the location can be incorporated directly in the code stream, with appropriate support from the linker.

Case 3: Variables bound to constants

If a variable is bound to a constant, references to it can be replaced with the constant (via constant propagation), and the binding can be eliminated, e.g.:

```
(let ([x 3])
  (letrec ([f (lambda () x)])
    —))
```

can be rewritten as:

```
(letrec ([f (lambda () 3)])
  —)
```

If this transformation is performed in concert with the other optimizations described in this section, a variable bound to a constant can be removed from the sets of free variables in which it appears.

Our compiler performs this sort of transformation prior to closure optimization, but this situation can also arise when a closure is allocated statically and treated as a constant by Case 2a of Section 5.2.1. For structured data, such as closures, care should also be taken to avoid replicating the actual structure when the variable is referenced at multiple points within its scope. Downstream passes of our compiler guarantee that this is the case, in cooperation with the linker, effectively turning the closure into a constant.

Case 4: Aliases

A similar transformation can take place when a variable x is bound directly to the value of another variable y, e.g.:

```
(let ([x y])
  (letrec ([f (lambda () x)])
    —))
```

can be rewritten (via copy propagation) as:

```
(letrec ([f (lambda () y)])
  —)
```

This transformation would not necessarily be valid if either x or y were assigned, but we assume that assignment conversion has already been performed.

In cases where both x and y are free within the same λ-expression, we can remove x and leave just y. For example, x and y both appear free in the λ-expression bound to f:

```
(let ([x y])
  (letrec ([f (lambda () (x y))])
    —))
```

Yet, if references to x are replaced with references to y, only y should be retained in the set of free variables.

Again, our compiler eliminates aliases such as this in a pass that runs before closure optimization. Nevertheless, this situation can arise as a result of Case 1b of Section 5.2.1, in which a closure for a well-known procedure with one free variable is replaced by its single free variable. It can also arise as the result of closure sharing, as discussed in Section 5.2.3

Case 5: Self-references

A procedure that recurs directly to itself through the name of the procedure has its own name as a free variable. For example, the λ-expression in the code for f below has f as a free variable:

```
(define append
  (lambda (ls1 ls2)
    (letrec ([f (lambda (ls1)
                  (if (null? ls1)
                      ls2
                      (cons (car ls1)
                            (f (cdr ls1) ls2))))])
      (f ls1))))
```

From the illustration of the closure in Figure 5.1, it is clear that this self-reference is unnecessary. If we already have f's closure in hand, there is no need to follow the indirect to find it. In general, a link at a known offset from the front of any data structure that always points back to itself is unnecessary and can be eliminated. Thus, a procedure's name need not appear in its own list of free variables.

Case 6: Unnecessary mutual references

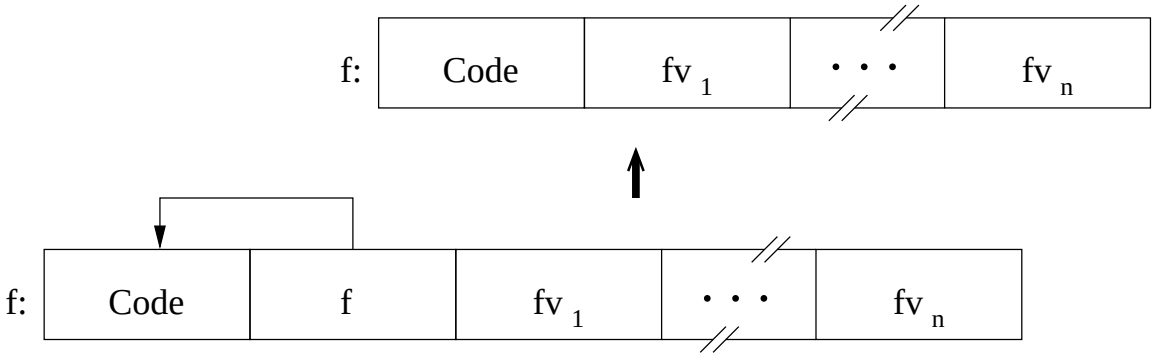

FIGURE 5.1. Function f with a self-reference in its closure.

A similar situation arises when two or more procedures are mutually recursive and have only the variables of one or more of the others as free variables. For example, in:

```
(letrec ([even? (lambda (x)
                  (or (= x 0)
                      (odd? (- x 1))))]
         [odd? (lambda (x) (not (even? x)))])
  —)
```

`even?` has `odd?` as a free variable only to provide `odd?` its closure and vice versa. Neither is necessary. This situation is illustrated in Figure 5.2.

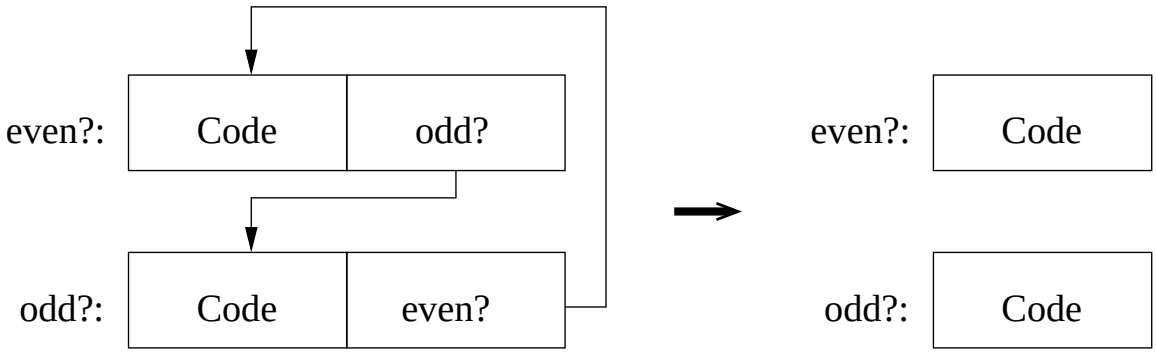

FIGURE 5.2. Mutual references for the `even?` and `odd?` closures.

In contrast, in the modified version below:

```
(lambda (z)
  (letrec ([even? (lambda (x)
                    (or (= x z)
                        (odd? (- x 1))))]
           [odd? (lambda (x) (not (even? x)))])
    —))
```

`z` is free in `even?`, so `even?` does need its closure to hold `z`, and `odd?` needs its closure to hold `even?`. This situation is illustrated in Figure 5.3.

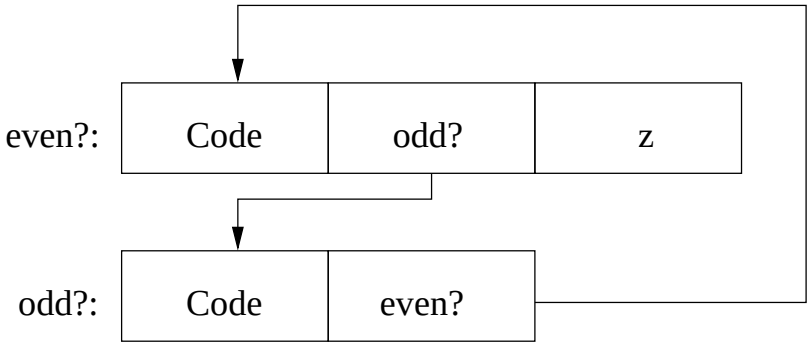

FIGURE 5.3. Mutual references for the `even?` and `odd?` closures, with `z` free in `even?`.

5.2.3. Sharing closures. If a set of closures cannot be eliminated, they possibly can be shared. For example, in the second `even?` and `odd?` example of Section 5.2.2, we could use a single closure for both `even?` and `odd?`. The combined closure would have just one free variable, `z`, as the pointer from `odd?` to `even?` would become a self-reference and, thus, be unnecessary. Further, when `even?` calls `odd?`, it would just pass along the shared closure rather than indirecting its own to obtain `odd?`'s closure. The same savings would occur when `odd?` calls `even?`.

There are three challenges, however. First, our representation of closures does not have space for multiple code pointers. This can be addressed with support from the storage manager, although not without some difficulty.

Second, more subtly, if two procedures have different lifetimes, some of the free-variable values might be retained longer than they should be. In other words, the representation is no longer "safe for space" [**78**]. This problem does not arise if either (a) the procedures have the same lifetime, or (b) the set of free variables (after removing mutually recursive references) is the same for all of the procedures.

Third, even more subtly, if two procedures have different lifetimes, but the same set of free variables, and one or more are not-well-known, one of the code pointers might be retained longer than necessary. In systems where all code is static, this is not a problem, but our compiler generates code on the fly, e.g., when the `eval` procedure is used; and anything that can be dynamically allocated must be subject to garbage collection, including code. This is not a problem when each of the procedures is well-known, assuming that we choose the vector representation over the closure representation in Case 1d of Section 5.2.1.

Thus, we can share closures in the following two cases:

Case 1: Same lifetime, single code pointer

Without extending our existing representation to handle multiple code pointers, we can use one closure for any set of procedures that have the same lifetime, as long as, at most, one of them requires its code pointer. Proving that two or more procedures have the same lifetime is difficult in general, but it is always the case for sets of procedures where a call from one can lead directly or indirectly to a call to each of the others, i.e., sets that are strongly connected [**86**] in a graph of bindings linked by free-variable relationships.

Case 2: Same free variables, no code pointers

If a set of well-known procedures all have the same set of free variables, the procedures can share the same closure, even when they are not part of the same strongly connected group of procedures. No harm is done if one outlasts the others, as the shared closure directly retains no more than what each of the original closures would have indirectly retained. In determining this, we can ignore variables that name members of the set, as these will be eliminated as self-references in the shared closure.

In either case, sharing can result in aliases that can lead to reductions in the sizes of other closures (Case 4 of Section 5.2.2).

5.2.4. Example. Consider the `letrec` expression in the following program:

```
(lambda (x)
  (letrec ([f (lambda (a) (a x))]
           [g (lambda () (f (h x)))]
           [h (lambda (z) (g))]
           [q (lambda (y) (+ (length y) 1))])
    (q (g))))
```

As the first step in the optimization process, we identify the free variables for the procedures defined in the `letrec`: x is free in f; x, f, and h are free in g; and g is free in h. q contains no free variables. We do not consider `+` or `length` to be free in q, as the locations of global variables are stored directly in the code stream, as discussed in Case 2 of Section 5.2.2. Additionally, we note that f, g, h, and q are all well-known.

Next, we partition the bindings into strongly connected components, producing one `letrec` expression for each [**56**,**93**]. g and h are mutually recursive, and, thus, must be bound by the same `letrec` expression, while f and q each get their own. Since f appears in g, the `letrec` that binds f must appear outside the `letrec` that binds g and h. Since q neither depends on nor appears in the other

procedures, we can place its `letrec` expression anywhere among the others. We arbitrarily choose to make it the outermost `letrec`.

After these partitions we have the following program:

```
(lambda (x)
  (letrec ([q (lambda (y) (+ (length y) 1))])
    (letrec ([f (lambda (a) (a x))])
      (letrec ([g (lambda () (f (h x)))]
               [h (lambda (z) (g))])
        (q (g))))))
```

We can now begin the process of applying optimizations. Since q is both well-known and has no free variables, its closure can be completely eliminated (Case 1a of Section 5.2.1). f is a well-known procedure and has only one free variable, x, so its closure is just x (Case 1b of Section 5.2.1). g and h are mutually recursive, so it is tempting to eliminate both closures, as described by Case 6 of Section 5.2.2. However, g still has x as a free variable, and, therefore, needs its closure. h also needs its closure so that it can hold g. Because g and h are well-known and are part of the same strongly connected component, they can share a closure (Case 1 of Section 5.2.3). Additionally, since f's closure has been replaced by x, there is only a single free variable, x, so the closures for g and h are also just x (Case 1b of Section 5.2.1). If another variable, y, were free in one of g or h, the result would be a shared closure represented by a pair of x and y (Case 1c of Section 5.2.1). If, further, g were not-well-known, a shared closure for g and h would have to be allocated with the code pointer for g and x and y as its free variables (Case 1 of Section 5.2.3).

5.3. The Algorithm

We now turn to a description of an algorithm that can be used to perform the optimizations described above. To simplify the presentation, we describe the algorithm by using the small core language defined in Figure 5.4. The grammar enforces a few preconditions on the input:

- variables are not assigned,
- `letrec` expressions are *pure*, i.e., bind (unassigned) variables to λ-expressions, and
- λ-expressions appear nowhere else,

The first precondition can be arranged via standard assignment conversion [**30**,**65**], while the second requires some form of `letrec` purification [**56**,**93**]. The third precondition can be arranged trivially via the following local transformation on λ-expressions.

$$\texttt{(lambda (}x\texttt{) }e\texttt{)} \rightarrow \texttt{(letrec ([}f'\texttt{ (lambda (}x\texttt{) }e\texttt{)]) }f'\texttt{)}$$

where f' is a fresh variable.

The algorithm requires one more precondition not enforced by the grammar:

- variables are uniquely named.

This precondition can be arranged via a simple alpha renaming.

In broad strokes, the closure optimization algorithm is as follows, with details provided in the referenced sections:

(1) Gather information about the input program, including the free variables of each λ-expression and whether each λ-expression is well-known (Section 5.3.1).
(2) Partition the bindings of each input `letrec` expression into separate sets of bindings known to have the same lifetimes, i.e., sets of strongly connected bindings (Section 5.3.2).
(3) When one or more bindings of a strongly connected set of bindings is well-known (i.e., they are bindings for well-known procedures), decide which should share a single closure (Section 5.3.3).
(4) Determine the required free variables for each closure, leaving out those that are unnecessary (Section 5.3.4).
(5) Select the appropriate representation for each closure and whether it can share space with a closure from some outer, strongly connected set of bindings (Section 5.3.5).
(6) Rebuild the code based on this selection (Section 5.3.6).

The final output of the algorithm is in the intermediate language shown in Figure 5.9 on page 134.

5.3.1. Gathering information. Before it can proceed, the main part of the algorithm requires a few pieces of information to be teased out of the program via static analysis:

- each λ-expression's free variables,
- call sites where the callee is known, and
- whether each λ-expression is well-known.

The set of free variables can be determined for each λ-expression via a straightforward recursive scan of the input program in which two values are returned at each step: (1) a new expression that records free variables at each λ-expression, and (2) the set of variables free in the expression. The base cases are variables and constants. The set of variables free in a variable reference includes just the variable itself, while the set of variables free in a constant is empty. The set of variables free in a `lambda`, `let`, or `letrec` is the union of the sets of variables free in each subform minus those

bound by the form.[10] The set of variables free in a `call` or primitive application is the union of the sets of variables free in the subforms. The result of this scan is the intermediate language shown in Figure 5.5, which differs from the core language only in the appearance of a free-variable set in the syntax for `lambda`. The free-variable set, *fvs*, is the set of variables free in the `lambda`.

Determining the known-call sites and whether each λ-expression is well-known is a bit trickier. The desired result is the intermediate language shown in Figure 5.6, which differs from the preceding language in the addition of a label, l, and a well-known flag, *wk*, to each `lambda`, along with the addition of a label or bottom, denoted by $l?$, to each call. Each label represents the entry point of one λ-expression. A label is recorded in a `call` only if we can prove that the corresponding `lambda` is the only one ever called (directly) by that call. Similarly, the well-known flag, *wk*, on a `lambda` is true only if we can prove that its name is used only at call sites where it is known, i.e., at call sites where the label is recorded.

A completely accurate determination is generally impossible, but it is straightforward to compute a conservative approximation efficiently as follows. First, create an environment that maps variables to labels and a separate store that maps variables to well-known flags, both initially empty. Then, for each `letrec`, create a fresh label for each of its λ-expressions; mark each variable well-known in the store; process its body and the bodies of each of its `lambda` subforms in an extended environment that maps each LHS variable to the corresponding label; and rebuild, using the processed subforms. For each of the rebuilt `lambda` subforms, record the corresponding label and the store's final value of the corresponding well-known flag. To process a `call` whose first subexpression is a variable that the environment maps to a label, process the second subexpression and rebuild the call with the label in the first position, followed by the variable and the processed second subexpression. To process any other call, process the subforms and rebuild with $\perp$ in the first position. To process a variable outside of the first position of a call, mark the variable not-well-known in the store. To process a `let` or primitive application, process the subforms and rebuild using the processed subforms. No processing is needed for constants. This process is linear in the size of the code.

The analysis above uncovers known calls only when the name of a λ-expression is in scope at the point of call. A more precise approximation can be made using a more elaborate form of control-flow analysis, as described by Serrano [**76**], or a type recovery that differentiates between individual procedures, as described by Adams et al. [**6**].

[10] For `(let (`x e_1`)` e_2`)`, x cannot appear free in e_1, as variables are uniquely named.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & (\texttt{let}\ (x\ e_1)\ e_2) \\
& | & (\texttt{letrec}\ ([x_1\ f_1]\ \ldots)\ e) \\
& | & (\texttt{call}\ e_0\ e_1) \\
& | & (\mathit{prim}\ e\ \ldots) \\
f & ::= & (\texttt{lambda}\ (x)\ e)
\end{array}
$$

$c \in \mathit{Const},\ x \in \mathit{Var},\ \mathit{prim} \in \mathit{Prim},$
$e \in \mathit{Exp},\ f \in \mathit{Fun}$

FIGURE 5.4. The core intermediate language.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & (\texttt{let}\ (x\ e_1)\ e_2) \\
& | & (\texttt{letrec}\ ([x_1\ f_1]\ \ldots)\ e) \\
& | & (\texttt{call}\ e_0\ e_1) \\
& | & (\mathit{prim}\ e\ \ldots) \\
f & ::= & (\texttt{lambda}\ \underline{\mathit{fvs}}\ (x)\ e)
\end{array}
$$

$\mathit{fvs} \in \mathcal{P}(\mathit{Var})$

FIGURE 5.5. Intermediate language after uncovering free variables.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & (\texttt{let}\ (x\ e_1)\ e_2) \\
& | & (\texttt{letrec}\ ([x_1\ f_1]\ \ldots)\ e) \\
& | & (\texttt{call}\ \underline{l?}\ e_0\ e_1) \\
& | & (\mathit{prim}\ e\ \ldots) \\
f & ::= & (\texttt{lambda}\ \underline{l}\ \underline{\mathit{wk}}\ \mathit{fvs}\ (x)\ e) \\
l? & ::= & l \mid \bot
\end{array}
$$

$l \in \mathit{Label},\ \mathit{wk} \in \mathit{Bool}$

FIGURE 5.6. Intermediate language after uncovering known calls.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & (\texttt{let}\ (x\ e_1)\ e_2) \\
& | & (\underline{\texttt{scletrec}}\ ([x_1\ f_1]\ \ldots)\ e) \\
& | & (\texttt{call}\ l?\ e_0\ e_1) \\
& | & (\mathit{prim}\ e\ \ldots) \\
f & ::= & (\texttt{lambda}\ l\ \mathit{wk}\ \mathit{fvs}\ (x)\ e) \\
l? & ::= & l \mid \bot
\end{array}
$$

FIGURE 5.7. Intermediate language after computing strongly connected sets.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & (\texttt{let}\ (x\ e_1)\ e_2) \\
& | & (\texttt{scletrec}\ (\underline{(}[x_1\ f_1]\ \ldots\underline{)}\ \ldots)\ e) \\
& | & (\texttt{call}\ l?\ e_0\ e_1) \\
& | & (\mathit{prim}\ e\ \ldots) \\
f & ::= & (\texttt{lambda}\ l\ \mathit{wk}\ \mathit{fvs}\ (x)\ e) \\
l? & ::= & l \mid \bot
\end{array}
$$

FIGURE 5.8. Intermediate language after choosing subsets for sharing.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & l \\
& | & (\texttt{let}\ (x\ e_1)\ e_2) \\
& | & \underline{(\texttt{labels}\ ([l_1\ f_1]\ \ldots)\ e)} \\
& | & (\texttt{call}\ l?\ e_0\underline{?}\ e_1) \\
& | & (\mathit{prim}\ e\ \ldots) \\
f & ::= & (\texttt{lambda}\ \underline{\mathit{cp}?}(x)\ e) \\
l? & ::= & l \mid \bot \\
e? & ::= & e \mid \bot \\
\mathit{cp}? & ::= & \mathit{cp} \mid \bot
\end{array}
$$

$\mathit{cp} \in \mathit{Var}$

FIGURE 5.9. Final output intermediate language.

5.3.2. Partitioning bindings into strongly connected sets. The next step in the algorithm is to determine sets of bindings that we can prove have the same lifetimes, as these are the ones that can potentially share closures without leading to space safety issues. As discussed in Section 5.2.3,

```
(make-closure code length)
(closure-set! closure index value)
(closure-ref closure index)
(vector e ...)
(vector-ref vector index)
(cons e1 e2)
(car pair)
(cdr pair)
```

FIGURE 5.10. Closure-related primitives.

proving that two or more procedures have the same lifetime is difficult in general, so we use a conservative approximation, which assumes the same lifetime only for members of each set of bindings that are strongly connected [**86**] in a graph of bindings linked by free-variable relationships. In determining strongly connected sets of `letrec` bindings, it is sufficient to consider each `letrec` form individually, as the bindings of two separate `letrec` forms can be connected via free-variable links in, at most, one direction.

Sets of strongly connected `letrec` bindings are also the minimal sets of closures that must be allocated and initialized together so that links can be established among them, as described in Section 5.3.6.

The result of the strongly connected component analysis is a program in the new intermediate language shown in Figure 5.7, which differs from the language in Figure 5.6 only in that `letrec` forms have been replaced with `scletrec` forms. In general, each `letrec` form is replaced by one or more `scletrec` forms nested so that if the λ-expressions of one `scletrec` reference bindings of another `scletrec`, the first is nested within the second.

If `letrec` purification has been performed via the algorithm described by Ghuloum and Dybvig [**56**], the `letrec` expressions already have been split into strongly connected sets of bindings, and this step need not be repeated.

5.3.3. Combining bindings. Once our `letrec` forms have been split into strongly connected sets represented by `scletrec` forms, we are ready to determine the subsets of each set that will share a single closure.

If our representation allowed closures to have multiple code slots, we could use a single closure for all of the bindings of the set. Because the representation permits us just one code slot, however, we must ensure that, at most, a single code pointer is needed by the members of each subset.

A code pointer is needed only for not-well-known bindings, as a well-known λ-expression is invoked via its direct-call label. Thus, if there are no not-well-known bindings in the set, no code pointer is required, and we can group all of the bindings together. This leads eventually to a run-time representation without any code pointer, as covered by Case 1 of Section 5.2.1.

If there are not-well-known bindings in the set, however, we must have at least one closure for each. There is no advantage to creating a separate closure for the well-known bindings, so we arrange for each of the well-known bindings to share a closure with one of the not-well-known bindings. Thus, we end up with exactly as many subsets as there are not-well-known bindings in the set.

When more than one not-well-known binding exists in a set, it is unclear how to distribute the well-known bindings among the resulting subsets. If possible, we would like to combine the not-well-known bindings with well-known bindings in a way that leads to minimizing the total number of free variables in each closure. However, this situation arises infrequently enough in the programs that we tested that we were not able to determine a generally useful heuristic, so our algorithm presently groups all of the well-known bindings in with an arbitrary pick of the not-well-known bindings. This transformation never does any harm, but it might not be as beneficial as some other combination in some circumstances.

The output of this step is in the intermediate language shown in Figure 5.8, in which the bindings of the `scletrec` are grouped into subsets that will share a single closure.

5.3.4. Determining required free variables. As discussed in Section 5.2.2, we need not include in our free-variable sets all of the variables that occur free within our λ-expressions. In fact, we must eliminate any `letrec`-bound variable that is neither bound nor referenced in the final output because its closure has been eliminated under Case 1a of Section 5.2.1. We can also eliminate global variables, variables bound to constants, aliases for other variables already included, self-references, and unnecessary mutual references.

To eliminate aliases and variables bound to constants, we use an environment that maps variables to expressions or $\perp$:

$$\rho \in Env = Var \rightarrow Exp \cup \{\perp\}$$

For the purpose of determining required free variables, the environment need map variables only to constants, other variables, and $\perp$; but we need other expressions when rebuilding the code, as described in Section 5.3.6.

If a local variable x is an alias for another variable y, ρ maps x to y. Similarly, if x is bound to a constant c, ρ maps x to c. If x is unbound in the final output, ρ maps x to $\perp$. Otherwise, ρ maps x to itself.

In our compiler, global variables are distinct from local variables and are referenced and assigned via static locations embedded in the code stream. Thus, they never appear in our free-variable lists and, hence, need not be eliminated. If this were not the case, we could recognize the case where ρ does not contain a mapping for a variable and treat it as global.

We construct the environment and determine the required free variables via an outermost to innermost traversal of the input program, starting with an empty environment and augmenting it when we encounter a `let` or `scletrec`. For `(let (`$x\ e_1$`)` e_2`)`, the environment ρ' used while processing e_2 is the result of augmenting ρ, as follows:

- if e_1 is a variable y, $\rho' x = \rho y$;
- if e_1 is a constant c, $\rho' x = c$;
- otherwise, $\rho' x = x$.

In the first case, ρ' maps x to ρy rather than to y because y might itself be an alias, bound to a constant, or unbound.

For `scletrec`, we select representations as described in the following section and augment the environment used while processing the `lambda` and `letrec` bodies, based on the representations selected, as follows.

- if a `letrec`-bound variable x is not needed because its closure has been eliminated, $\rho' x = \perp$;
- if the closure for x is a constant closure c, i.e., one containing just a code pointer, $\rho' x = c$;
- if the closure for x is just the value of the free variable y, $\rho' x = y$;
- similarly, if the closure (or other data structure) for x is shared with a closure to which another variable y has been bound, $\rho' x = y$;
- otherwise, $\rho' x = x$.

Because the traversal proceeds from outermost to innermost, by the time it is ready to process a given `scletrec` form, information about the free variables of its λ-expressions, except those bound by the `scletrec` form itself, is available in the environment. We optimistically assume that none of the variables bound by the `scletrec` form is needed until we have proved otherwise. Thus, we compute initial free-variable sets for each group of bindings based only on free variables that are not bound by the current `scletrec` form, using the environment to eliminate unbound variables, constants, and aliases.

To be precise, if a variable $x \in fvs$ for any *fvs* among the λ-expressions of one group of bindings of an `scletrec`, ρx is included in the initial free-variable set of the group if and only if ρx is a variable. This effectively omits unbound variables and variables bound to constants. It also eliminates aliases because, if the environment maps x and y to z (where z might be x, y, or some variable distinct from x and y), only z appears in the resulting set.

We must still decide which of the variables bound by the current `scletrec` must be included in the free-variable sets of each group. At most, one variable bound by each group needs to be included, as each of the variables will be bound to the same closure (if any). Thus, we pick an arbitrary representative variable *rep* from the set of variables bound by each group and decide whether to include it in the free-variable sets of the others.

We never include *rep* in the free-variable set of its own group, effectively eliminating self-references (Case 5 of Section 5.2.2). If all of the initial free-variable sets are empty, no representatives are added, and the final free-variable sets are also empty. This effectively eliminates unnecessary mutual references (Case 5 of Section 5.2.2).

If, however, the free-variable set of any group is non-empty, we must add the representative *rep* of each group G to the free-variable set of each other group H if G has a non-empty set of free variables and $rep \in fvs$ for some *fvs* among the λ-expressions of H. Each group must have at least one of the other representatives in one or more *fvs*, so in this case, each of the final free-variable sets is non-empty.

The products of this step are the final free-variable sets, one for each group of bindings of a single `scletrec` form. The selected representative *rep* for each group must also be communicated to the next step.

5.3.5. Selecting representations. Once we have the final set of free variables for each group, we are ready to decide how each group's closure is represented. This step is performed for each `scletrec` during the outermost to innermost traversal of the program, described in Section 5.3.4. For each group in an `scletrec`, we select the representation determined by the cases in Section 5.2.1 as follows:

Case 1a: Well-known with no free variables

Because we have opted to combine all well-known bindings with a not-well-known binding if one exists, this case and the other well-known cases occur only if there is a single group whose λ-expressions are all well-known.

In this case, no closure is needed, and ρ' maps *rep* and the other variables bound by the group to $\perp$.

Case 1b: Well-known with one free variable x

In this case, the closure is just the value of x, and ρ' maps *rep* and the other variables bound by the group to x.

Case 1c: Well-known with two free variables x and y

In this case, the closure is a run-time allocated pair that contains the values of x and y. ρ' maps each of the variables bound by the group to *rep*, and code to create the pair and bind *rep* to the pair is generated as described in Section 5.3.6.

Case 1d: Well-known with three or more free variables x ...

In this case, the closure is a run-time allocated vector that contains the values of x .... ρ' maps each of the variables bound by the group to *rep*, and code to create the vector and bind *rep* to the vector is generated as described in Section 5.3.6.

Case 2a: Not-well-known with no free variables

In this case, the closure is a constant closure c that contains just a code pointer, and ρ' maps *rep* and the other variables bound by the group to c.

Case 2b: Not-well-known with one or more free variables x ...

In this case, the closure is a run-time allocated closure whose code pointer is the label of the single not-well-known binding in the group, and its free-variable slots hold the values of x .... ρ' maps each of the variables bound by the group to *rep*, and code to produce the closure and bind *rep* to the closure is generated, as described in Section 5.3.6.

In cases 1c and 1d, we have a further opportunity to share closures, which is to locate a binding created by an outer `scletrec` that would have exactly the same set of free variables if the pair or vector were shared. For example, an outer closure might be represented as a pair with free variables x and y. If the new closure also has just x and y as free variables, there is no harm in borrowing the pair used for the outer closure. Further, if the new closure has f as a free variable as well as x and y, it can still use the outer pair, as f becomes a self-reference and, hence, is not needed.

To implement this form of sharing, the algorithm maintains an additional compile-time environment, *bank*, that maps sets of free variables to the representative variables of closures available to be borrowed. We deposit into *bank* only closures represented as pairs or vectors, as we cannot borrow

a closure with a code pointer without possibly retaining the code pointer too long, as described in Section 5.2.3. For any well-known procedure with more than one free variable, we check to see whether a closure with a compatible set of free variables already exists in *bank* and, if so, map *rep* and the other variables bound by the group to the representative variable of the borrowed closure. While processing the body of a λ-expression, we withdraw from *bank* those closures whose names do not appear free in the λ-expression, as they are not visible in the body.

5.3.6. Rebuilding the code. The final step of the algorithm is to produce the output code. This step is performed for each `scletrec` during the traversal of the code described in Section 5.3.4, based on the decisions made in Section 5.3.5. The intermediate language for the final output is shown in Figure 5.9 and may require some or all of the primitive operations shown in Figure 5.10. Producing the output code involves:

- generating a `labels` form to bind labels to lambda expressions,
- adding a closure-pointer (*cp*) variable to each λ-expression that requires its closure,
- rewriting variable references, and
- generating code to create the required pairs, vectors, and closures.

The generated `labels` form binds the label associated with each λ-expression in each group of the `scletrec` form to the corresponding λ-expression. Although the `labels` form appears where the `scletrec` form originally appeared, the scope of each label is the entire input program, so that any call can jump directly to the code produced by the corresponding λ-expression. In fact, because the resulting λ-expressions access their free variables, if any, through an explicit closure argument, all `labels` forms and the λ-expressions within them can be moved to the top level of the program as part of this or some later transformation, if desired.

If a λ-expression has any free variables, it is given a closure-pointer (*cp*) variable; when code is ultimately generated for the λ-expression, *cp* must be assigned to the incoming closure (or pair or vector), just as the formal parameter x is assigned to the incoming actual parameter. In essence, *cp* is merely an additional formal parameter, and the closure is merely an additional actual parameter.

Each reference to a free variable within a λ-expression must be replaced by an expression to retrieve the value of the free variable from the closure. Similarly, a reference to a variable bound to a constant must be replaced by the constant, a reference to a variable bound to another variable must be replaced by a reference to the other variable, and references to unbound variables must be eliminated. Sufficient information already exists in the incoming environment ρ to handle the latter set of cases, where each reference to a variable x is simply replaced during the outermost to

innermost traversal by ρx. Unbound variables arise only from well-known procedures and can thus appear only in the second (closure) position of a `call` and only when the `call`'s label is not $\perp$. When this happens, the variable maps to $\perp$, and the reference to the variable is replaced in the `call` by $\perp$, which frees the caller from passing any sort of closure to the callee.

To handle free variables, ρ is augmented to create an environment ρ', as follows:

- for a self-reference x, $\rho' x = cp$;
- for a procedure represented by one of its free variables x, $\rho' x = cp$;
- for a procedure represented by a pair of x and y, $\rho' x =$ `(car` cp`)`, and $\rho' y =$ `(cdr` cp`)`;
- for a procedure represented by a vector of $x_1 \ldots x_n$, $\rho' x_i =$ `(vector-ref` cp i`)` for $0 \leq i < n$;
- for a procedure represented by a closure of $x_1 \ldots x_n$, $\rho' x_i =$ `(closure-ref` cp i`)` for $0 \leq i < n$.

The augmented environment ρ' is used while processing the body of each `lambda` form as well as the body of the `scletrec` form, as references to the `letrec`-bound variables can occur in both contexts.

Generating code to create the required pairs, vectors, and closures is straightforward, with one minor twist. Because the links among the non-constant closures created for the set of groups in an `scletrec` necessarily form one or more cycles, the code to create them allocates all of the closures before storing the values of the free variables. For example, if an `scletrec` form with body e has two groups, one with representative r_1, label l_1, and free variables r_2 and y; and one with representative r_2, label l_2, and free variable r_1; the `scletrec` form is replaced with the following:

```
(let ([r1 (make-closure l1 2)] [r2 (make-closure l2 1)])
  (closure-set! r1 0 r2)
  (closure-set! r1 1 y)
  (closure-set! r2 0 r1)
  e)
```

The order of the bindings and the order of the `closure-set!` forms do not matter.

5.4. Results

Our implementation extends the algorithm described in Section 5.3 to support the full R6RS Scheme language [**83**]. To determine the effectiveness of closure optimization, we ran the optimization over a standard set of 67 R6RS benchmarks [**26**] and instrumented the compiler and resulting machine code to determine:

- statically,
  - the number of closures eliminated and

  - the reduction in the total number of free variables; and
- dynamically,
  - the reduction in allocation cost and
  - the reduction in the number of memory references.

Overall, the optimization performs well, on average statically eliminating 56.94% of closures and 44.89% of the total free variables, and dynamically eliminating, on average, 58.25% of the allocation and 58.58% of the memory references attributable to closure access.[11]

These numbers are gathered after a pass that performs aggressive inlining, constant propagation, constant folding, and copy propagation [**90**]. It also follows a pass that recognizes loops and converts them into the equivalent of labels and gotos. As a result, the numbers do not include the benefits of eliminating variables bound to constants or other variables, except where these situations arise during closure optimization. Inlining and loop recognition also eliminate the need for some closures and can affect the number of free variables (potentially increasing this number in some cases and decreasing it in others). Global variables are not counted as free variables, as they are stored at a fixed location and accessed via primitives that set or retrieve their values.

Constant closures and those replaced by a single free variable are considered eliminated in our numbers, as neither incurs run-time overhead. In counting the allocation of vectors and closures, we include the space required for the length (in the case of vectors) and the code pointer (in the case of closures), as well as the space required to hold the free-variable values.[12]

In addition to the overall numbers, we ran, in isolation, optimizations that eliminate self-references, eliminate mutual references, share closures in strongly connected sets of bindings, borrow closures from outer sets of bindings, and select more efficient representations. Table 5.1 shows the breakdown in the percentage of eliminated closures, free variables, memory references, and allocation by optimization. The table shows that running the optimizations together results in greater benefits than does running the optimizations separately. This is because some of the optimizations can lead to opportunities for the others.

The results for the complete set of benchmarks are impressive but vary from benchmark to benchmark. Some of the benchmarks are simpler benchmarks for running functions such as factorial, Tak, and Fibonacci. It is interesting to see how much these benchmarks benefit from the closure

[11]The percentage of memory stores attributable to closure initialization also decreases in direct proportion with the reduction in allocation.
[12]The numbers do not include pad words required to maintain object alignment, e.g., double-word alignment on 32-bit machines.

TABLE 5.1. Eliminated closures (Closure), free-variables (FV), memory references (Mem. Ref.), and allocation (Alloc.) by closure optimization type.

Optimization	**Closure**	**FV**	**Mem. Ref.**	**Alloc.**
Self-ref.	0.00%	25.41%	45.64%	19.33%
Mutual-ref.	0.00%	7.91%	32.55%	6.14%
Representation	29.65%	3.48%	1.23%	20.78%
Sharing	1.91%	3.17%	0.00%	0.58%
Borrowing	0.20%	0.28%	0.00%	0.02%
All	56.94%	44.89%	58.58%	58.25%

optimization, but, ultimately, the larger programs within the benchmark suite help to provide a better indicator of how well these optimizations will work on real-world programs. The R6RS "compiler" benchmark is the largest in terms of source code and has the most closures initially. Statically, 40.85% of closures and 31.9% of free variables are eliminated, and, dynamically, 31.52% of the allocation and 27.59% of the memory references attributable to closure creation and access are eliminated. This example might be more typical of the average real-world program.

The R6RS "simplex" benchmark is an example of a benchmark that does not benefit much from our closure optimization, eliminating, statically, only 8.00% of the closures and 14.97% of free variables, while eliminating, dynamically, 13.96% of the allocation and 9.84% of the memory references attributable to closure creation and access. The benchmark is written as a set of functions that mutate data structures pointed to from free variables. Because most of the free variables are not other closures (and some that are have already been inlined by the source optimization pass), there are not many closures that can be eliminated.

The R6RS "nucleic" benchmark is an example of a longer benchmark that performs better than average, eliminating 67.74% of the closures and 66.23% of the free variables statically, and 37.43% of the allocation and 72.69% of the memory references attributable to closure creation and access at run time. This benchmark has many top-level definitions that are either λ-expressions or constants, so the free variables tend to be other closures. Although data structures are also mutated here, they tend to be passed as arguments rather than stored as free variables in the closure. In a program such as this, we expect to see many of the closures eliminated.

In addition to testing the performance of the benchmarks, we also measured the effectiveness of the optimization algorithm when run on the sources for our own compiler. Statically, 45.67% of closures and 32.36% of free variables are eliminated, and, dynamically, 47.52% of the allocation and 47.00% of the memory references attributable to closure creation and access are eliminated.

We also measured how our optimizations affected the run times of our benchmarks. The decrease in run times ranged from negligible up to 20%, with an average decrease of 3.6%. Run times actually increased for a few of the benchmarks. Because our optimizations are supposedly guaranteed to not add overhead, we examined one of those whose run time increased and determined that its poor performance was due to bad caching. Rearranging the code led to equivalent performance between the optimized and unoptimized versions of the code. Several of the benchmarks spend most of their time in procedures recognized as loops earlier and do not benefit at all from the closure optimization. Larger programs tended to experience greater improvement in run time, which correlates well with the other measurements. Memory allocation in our implementation is fast, averaging around three instructions, plus one store to initialize each field. For implementations with slower or even out-of-line allocation, the decrease in run time due to reduction in closure allocation would be greater. Similarly, implementations that do not perform inlining or loop recognition would likely benefit more from the optimizations. In comparison, systems with higher overhead in other places would likely benefit less.

Case 1 of Section 5.2.3 presents the possibility of extending our implementation to support multiple code pointers so that all of the bindings of a strongly connected set of bindings can share the same closure, even if more than one is not well-known. Because making this change to our system would be a major undertaking, we decided to determine the potential benefit of the optimization before proceeding. Our test showed that this optimization would affect less than one-tenth of one percent of `letrec` bindings, which led us to abandon the idea.

Our implementation of the optimization algorithm employs standard techniques to avoid the implied overhead of creating and updating the sets required by the algorithm. For example, it associates a *seen* flag with each variable to indicate when a free variable has already been added to a set. Although we have yet to create a proof, we believe that the implementation is linear in the number of variables free in all λ-expressions in the program, which is the best that any implementation of the flat-closure model can achieve.[13] In other words, our implementation adds, at most, constant overhead to the naive flat-closure model and can sometimes improve the speed of downstream passes via the elimination of closure operations. Indeed, the optimization adds essentially no measurable compile-time overhead in our compiler, with compile times varying by an average of less than 1% with the optimization disabled or enabled.

[13]In the worst case, the number of such free variables is quadratic in the size of the program [**78**], although the worst case appears to be approached rarely in practice.

5.5. Related Work

Our replacement of a well-known closure with a single free variable is a degenerate form of lambda lifting [**62**], in which each of the free variables of a procedure are converted into separate arguments. Increasing the number of arguments can lead to additional stack traffic, particularly for non-tail-recursive routines, and it can increase register pressure whenever two or more variables are live in place of the original single package (closure) with two or more slots. Limiting our algorithm to doing this replacement only in the single-variable case never does any harm, as we are replacing a single package of values with just one value.

Serrano [**76**] describes a closure optimization based on control-flow analysis [**80**]. His optimization eliminates the code part of a closure when the closure is well-known; in this, our optimizations overlap, although our benefit is less, as the code part of a closure in his implementation occupies four words, while ours occupies just one. He also performs lambda lifting when the closure is well-known and its binding is in scope wherever it is called.

Steckler and Wand [**84**] describe a closure-conversion algorithm that creates "light-weight closures" that do not contain free variables that are available at the call site. This is a limited form of lambda lifting and, as with full lambda lifting, can sometimes do harm relative to the straight flat-closure model.

Kranz [**65**] describes various mechanisms for reducing closure allocation and access costs, including allocating closures on the stack and allocating closures in registers. The former is useful for closures created to represent continuations in an implementation that uses a continuation-passing style [**57**] and achieves part of the benefit of the natural reuse of stack frames in a direct-style implementation. The latter is useful for procedures that act as loops and reduces the need to handle loops explicitly in the compiler. Our optimizations are orthogonal to these optimizations, but they do overlap somewhat in their benefits.

Shao and Appel [**78**] describe a nested representation of closures that can reduce the amount of storage required for a set of closures that share some but not all free variables, while maintaining space safety. The sharing never results in more than one level of indirection to obtain the value of a free variable. Because a substantial portion of the savings reported resulted from global variables [**9**], which we omit entirely, and we operate under the assumption that free-variable references are typically far more common than closure creation, we have chosen to stick with the flat-closure model and focus instead on optimizing that model.

Fradet and Métayer [**46**] describe various optimizations for implementations of lazy languages. They discuss reducing the size of a closure by omitting portions of the environment not needed by a procedure, which is an inherent feature of the flat-closure model preserved by our mechanism. They also discuss avoiding the creation of multiple closures when expressions are deferred by the lazy-evaluation mechanism in cases where a closure's environment, or portions of it, can be reused when the evaluation of one expression provably precedes another, i.e., when the lifetime of one closure ends before the lifetime of another begins.

Dragoş [**29**] describes a set of optimizations aimed at reducing the overhead of higher-order functions in Scala. A closure elimination optimization is included that attempts to determine when free variables are available at the call site or on the stack to avoid creating a larger class structure around the function. The optimization also looks for heap-allocated free variables that are reachable from local variables or the stack to avoid adding them to the closure. The optimization helps eliminate the closures for well-known calls by lambda lifting, if possible.

Appel [**11**] describes eliminating self-references and allowing mutually recursive functions (strongly connected sets of `letrec` bindings) to share a single closure with multiple code pointers. These optimizations are similar to our elimination of self-references and sharing of well-known closures, although in our optimization we allow only one not-well-known closure in a shared closure.

A few of the optimizations described in this chapter have been performed by Chez Scheme since 1992: elimination of self-references, elimination of mutual references where legitimate, and allocation of constant closures (though without the propagation of those constants). Additionally, we have seen references, in various newsgroups and blogs, to the existence of similar optimizations. While other systems may implement some of the optimizations that we describe, there is no mention of them, or an algorithm to implement them, in the literature.

5.6. Conclusion

The flat-closure model is a simple and efficient representation for procedures that allows the values or locations of free variables to be accessed with a single memory reference. This chapter presented a set of flat-closure compiler optimizations and an algorithm for implementing them. Together, the optimizations result in an average reduction in run-time closure-creation and free-variable access overhead on a set of standard benchmarks by over 50%, with insignificant compile-time overhead. The optimizations never add overhead, so a programmer can safely assume that a program will perform at least as well with the optimizations as with a naive implementation of flat closures.

CHAPTER 6

Conclusion and Future Work

When a micropass compiler was first conceived as a teaching aid in a compiler class, the idea of a compiler as a set of small, single-task passes was seen as too slow for use in a commercial compiler. Indeed, the class compiler, which handles a small subset of Scheme, slows down significantly as the number of passes increases. This slow down results from the overhead of matching and rebuilding the S-expressions used to represent the program being compiled. A compiler can take a noticeable amount of time to complete, even when compiling relatively small programs. Passes in the class compiler do not check that their output is well-formed, which creates the potential for a pass to generate a malformed intermediate representation that is not caught until a later pass. Passes also contain boilerplate code to traverse forms that do not otherwise change in the pass, burdening the student with the need to write more code.

The prototype nanopass framework developed by Sarkar demonstrated that it was possible to improve on this model, while maintaining the simplicity of working with an S-expression syntax. Her dissertation demonstrated that the `define-language` and `define-pass` forms could be used in concert to create more efficient passes that use less code. The passes also have the benefit of verifying that the output of each pass is a valid language form, without the need for any additional verification pass. The prototype is not up to the task of writing a commercial compiler. Nevertheless, the prototype laid the important groundwork that made this dissertation possible.

We set out with the goal of demonstrating that a suitably improved nanopass framework could be used to build a commercial compiler by building a new compiler that is compatible with the commercial Chez Scheme compiler, that generates code that is on par with that compiler, and runs within a factor of two of that compiler. The extra compile time reflects the fact that the original compiler is almost *absurdly* fast and allows us to experiment with a substantially slower graph-coloring register allocator. This goal has been achieved. The new compiler is compatible with the existing compiler, demonstrated by the fact that it passes the extensive test suite used to test Chez Scheme. A set of benchmarks performs better, on average, when compiled with the new compiler than with the original compiler, between 15.0%–26.6% faster depending on architecture

and optimization level. These numbers include two benchmarks that make use of the new, slower compiler at run time, which negatively affects the average time. The average compile time of the benchmarks is well within the factor of two, ranging from a factor of 1.64 on the 32-bit version at optimize level 3 to 1.75 on the 64-bit version at optimize level 2.

The nanopass framework, in our experience, made developing the new compiler an easier task. Adding a new form to a language that would last for several passes was made easier by the fact that the boilerplate code for it could often be autogenerated in the intervening passes. Even in those passes where the clause could not be autogenerated, it was often possible to predict where and how the new form would need to be handled by looking at how a similar form was handled. Over the course of the new compiler's development, we decided to change the order in which the tasks in the compiler were performed on several occasions. These changes were not necessarily trivial but were often simplified by the nanopass framework. When passes were moved, the source (and often target) language of the pass changed. When this changed in ways that affected the passes, problems caused by the change were often caught at expand time, when the forms of a pattern or template could not be met, or early in the run-time testing, when a pass received a language form from the wrong language. The improved error messages in the new nanopass framework helped to make these problems easier to find and fix.

The `library-group` form, described in Chapter 4, was developed before the front-end passes were rewritten to use the nanopass framework. Because earlier versions of the new compiler used the same front-end code, this remained unchanged until recently, when the expander was rewritten to use a nanopass language as the first intermediate representation. As the compiler processes the `library-group` form, it traverses the internal representation of the combined library form, replacing references to library globals for libraries included in the library group with local references. The original pass performing this traversal listed all of the forms used in the internal representation at that point in the compiler, but the rewritten pass only needs to look for the one form that changes and requires only one clause, with the other clauses being autogenerated. The original pass also had a bug resulting from the addition of forms when the implicit cross-library optimization was added. Now that the pass is implemented using the nanopass framework, the autogeneration of clauses ensures that changes to the intermediate language will not result in bugs.

The closure optimization, described in Chapter 5, is a good example of how causing a pass to perform a single task can make implementing the pass simpler and change the way the compiler writer thinks about the pass. When we started writing the closure optimization, our intention was to support

exactly the same features as those of the original compiler. In the original compiler, the work of closure optimization is spread among several passes, where the individual tasks are intermixed with several other optimizations and other transformations. Looking at the pass as a whole enabled us to start to pick away at the algorithm, both simplifying it and finding new opportunities for optimization. The mechanisms for combining multiple closures with the same lifetime into a single closure and borrowing closures created for another well-known procedure when the free variables are the same in both came out of this process. Combining closures simplified the task of removing mutual references from the free-variable lists.

No sophisticated tool is ever quite done. The more we have used and improved the nanopass framework, the more places we have found where the nanopass framework should be able to intuit the programmer's intentions and where more expressive power could allow the nanopass framework to further simplify the implementation of passes. With that said, however, the current version of the nanopass framework stands up to the task that we originally intended for it. It allowed us to build a new compiler, in roughly a year and a half, that is fully compatible with the existing Chez Scheme compiler. The new compiler is easier to work with because the intermediate languages used in the compiler are formally specified, and the task of each pass is clear. Adding new optimizations is simpler, as it is no longer necessary to try to weave them into existing passes that are already performing several other tasks.

Although the nanopass framework described in this dissertation is capable of being used in a commercial compiler, there are still improvements to be made, in terms of both performance and functionality. The section below presents future directions for this work.

Better determination of when a pattern is covered. In the current version of the nanopass framework, a production of a nonterminal is considered covered if a user-supplied clause matches the production pattern, without restrictions on any of the elements of the pattern and without a guard clause. This is sufficient in the vast majority of cases, but some intermediate languages contain a nonterminal with a single production, and it is often useful to match this single production as part of a larger pattern that contains the production. In these cases, `define-pass` generates a clause to handle the "most general" case. The clause is not reachable, as the user-supplied clause matches all possible forms, but it can cause `define-pass` to generate an entire shadow set of unreachable transformers. By checking nonterminal productions to see whether all cases are met before autogenerating a clause, generating this unreachable code could be avoided. In cases where

only a single production is valid as a sub-form, it would is also possible to avoid performing the test to make sure this form is found, since it is the only form that is possible.

Better autogeneration of transformers. Transformer autogeneration is intended to remove more of the boilerplate code from pass definitions and to allow the compiler writer to focus on the forms that need to be transformed. Unfortunately, transformer autogeneration can also lead to unexpected results. This can happen because the compiler writer missed a production in a nonterminal that needed to be handled, supplied an incorrect number of bindings in a catamorphism, or wrote an intermediate transformer that did not include all of the extra formals or return values needed for autogeneration of a recursive clause. This kind of bug can be difficult to track down, as it tends to raise errors at the run time of the compiler and not during the expansion of the compiler. The autogeneration of transformers is also limited to constructing transformers that take an input-language form as the only argument and produce only an output-language form. This avoids autogenerating a transformer that would have difficulty knowing how to handle extra formals or automatically generating extra return values. In developing the new Chez Scheme compiler, we found a desire for both more control of autogeneration for the user and a more general heuristic for determining when it is appropriate to autogenerate a transformer and when it is not.

To address the problem of autogenerated clauses appearing when they are not wanted, an optional clause could be added to `define-pass` to disable transformer autogeneration when it is not desired. This could be helpful as a debugging device, allowing the compiler writer to see missing transformers, and, more generally, when transformers are generated as the result of a pattern that does not seem to be matched, even though the user-supplied clauses had covered all of the productions.

To provide a more general autogeneration heuristic, autogenerated transformers could be allowed to take extra formals, particularly when there is a nonterminal production with a user-supplied transformer that expects the extra formal. Transformers that return no values could also be autogenerated to traverse the full language term when an effect-only transformer exists, without requiring the compiler writer to explicitly specify the recursion.

Polymorphic languages. Each language defined by `define-language` generates a distinct record for each S-expression production of each nonterminal. This means that no two languages ever share the same record types, even though they may share the same S-expression production in the same nonterminal. As a result, every pass in the compiler that uses different input and output languages must rewrite every nonterminal production in the language to produce a well-formed output. This is good for the checking that it provides, but, in some cases, particularly those where the majority

of the forms in the language do not need to be rewritten, such as the frame allocation and register allocation passes of the class compiler, this rewriting can lead to poor performance both due to the extra allocation of the new forms and due to the need to perform the work of building them.

One way around this is to allow languages with similar forms to use the same record to represent those forms. This also requires changes to the `define-pass` implementation, which would need to avoid generating clauses that traverse forms that do not need to be rewritten. This is also a feature that should be under the compiler writer's control, as, in some cases, it might be desirable to ensure a complete rewrite of the language term to provide the same sort of checking that exists in the current nanopass framework.

Flow-sensitive languages and passes. Automatically generated clauses in the nanopass framework visit the components of a language term in a flow-insensitive manner. This means that the extra return values from recursive calls on a sub-form will be lost in an autogenerated clause. It also means that if a formal is updated based on the output of a pass, for instance, an environment is updated while processing a sub-form, this, too, will be lost. Hence, in passes that require flow-sensitive traversal of language terms, the compiler writer is responsible for specifying the order of the traversal by writing clauses that match each language form that contains sub-forms that are affected by the flow-sensitive traversal.

In developing the new Chez Scheme compiler, we found that this led to many boilerplate clauses that simply recur on their sub-forms in the order required by the traversal and rebuild a matching output term. Avoiding this kind of boilerplate clause is one of the intentions of the nanopass framework, and by informing the `define-pass` clause of the traversal order, it should be possible to automatically generate these clauses as well. This is not as simple as specifying a tree traversal, however, because some of the intermediate languages in Chez Scheme include labels and gotos. This means that the control flow of a program is not structured as a tree, or even a DAG, but might contain cycles. This requires that the language form itself carries some information about how control flow is handled as a means to give `define-pass` the information that it needs to autogenerate clauses and processors.

Pass fusing. While the compile time of many compilers is driven by individual passes with high computational complexity, such as live analysis, having many individual passes does add some constant overhead. One way to mitigate this, while still using the nanopass approach, is to fuse multiple passes together when the passes can be run simultaneously. This avoids the creation of intermediate language representations and executes several passes in a single walk over the intermediate program term.

There are a number of challenges in fusing passes. The nanopass framework does not restrict the language that can be used in the right-hand side of a transformer clause, so compiler writers can use side-effecting operations that could lead to semantic changes when two passes are fused. This means that some non-trivial analysis is needed to determine when two passes can be fused. This can be difficult in the expander, as it requires a code walk. Even with the code walk, the expander would need to verify each identifier referenced in the passes to be fused to ensure no side-effects could occur that would prevent combining the passes.

Another option is to allow the compiler writer to specify when two passes should be fused. This removes the burden of analyzing the passes from the nanopass framework (and imposes it on the compiler writer). Even this is challenging, however. Pass fusion is a form of deforestation [**94**] where passes are composed to eliminate creation of intermediate language terms. Because the nanopass framework allows language terms to be constructed outside of the right-hand side of a pass, it might not be possible to determine, in all cases, what output term will be constructed as the output of the pass. Given these caveats, it is still possible that pass fusion could be useful. This might be implemented as an additional macro or as an optional instruction to the `define-pass` form that indicates that a given set of passes is part of the same fused pass.

Bibliography

[1] The Glasgow Haskell Compiler. URL `http://www.haskell.org/ghc/`.

[2] Scheme Libraries. URL `http://launchpad.net/scheme-libraries`.

[3] *ISO/IEC 14882:2003: Programming languages: C++*. 2003. URL `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110`.

[4] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN 0321227255.

[5] M. D. Adams and R. K. Dybvig. Efficient nondestructive equality checking for trees and graphs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 179–188. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-919-7. URL `http://doi.acm.org/10.1145/1411204.1411230`.

[6] M. D. Adams, A. W. Keep, J. Midtgaard, M. Might, A. Chauhan, and R. K. Dybvig. Flow-sensitive type recovery in linear-log time. In *OOPSLA*, pages 483–498, 2011.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Pub. Co., Reading, Massachusetts, 2nd edition, 2006.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter Introduction, pages 20–21. Addison-Wesley Pub. Co., Reading, Massachusetts, 1986.

[9] A. Appel. Private communication. 1994.

[10] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 40 West 20th Street, New York, NY 10011-4211, 1992.

[11] A. W. Appel. *Compiling with Continuations*, chapter Closure conversion, pages 103–124. Cambridge University Press, 40 West 20th Street, New York, NY 10011-4211, 1992.

[12] J. M. Ashley and R. K. Dybvig. An efficient implementation of multiple return values in Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 140–149. ACM, New York, NY, USA, 1994. ISBN 0-89791-643-3. URL `http://doi.acm.org/10.1145/182409.156784`.

[13] J. Bachrach. D-Expressions: Lisp power, Dylan style, 1999. URL `http://people.csail.mit.edu/jrb/Projects/dexprs.pdf`.

[14] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Proceedings of the third International Workshop on Source Code Analysis and Manipulation*, SCAM '03, pages 65–75. IEEE Computer Society Press, Amsterdam, The Netherlands, Sept. 2003. URL `http://www.codeboost.org/papers/codeboost-scam03.pdf`.

[15] D. Boucher. GOld: a link-time optimizer for Scheme. In *Proceedings of the 2000 Workshop on Scheme and Functional Programming*, Scheme '00, 2000.

[16] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52–70, June 2008. URL `http://dx.doi.org/10.1016/j.scico.2007.11.003`. Special issue on experimental software and toolkits.

[17] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16:428–455, May 1994. ISSN 0164-0925. URL `http://doi.acm.org/10.1145/177492.177575`.

[18] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 99–107. ACM, New York, NY, USA, 1996. ISBN 0-89791-795-2. URL `http://doi.acm.org/10.1145/231379.231395`.

[19] R. G. Burger and R. K. Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 108–116. ACM, New York, NY, USA, 1996. ISBN 0-89791-795-2. URL `http://doi.acm.org/10.1145/231379.231397`.

[20] R. G. Burger and R. K. Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proceedings of the 1998 International Conference on Computer Languages*, ICCL '98, pages 240–. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-8186-8454-2. URL `http://dl.acm.org/citation.cfm?id=857172.857260`.

[21] R. G. Burger, O. Waddell, and R. K. Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 130–138. ACM, New York, NY, USA, 1995. ISBN 0-89791-697-2. URL `http://doi.acm.org/10.1145/207110.207125`.

[22] L. Cardelli. The functional abstract machine. Technical report, Bell Laboratories, 1983.

[23] E. Cavazos. Dharmalab git repository. URL `http://github.com/dharmatech/dharmalab/tree/master/indexable-sequence/`.

[24] E. Cavazos. MPL git repository. URL `http://github.com/dharmatech/mpl`.

[25] W. D. Clinger, 2008. The Larceny Project.

[26] W. D. Clinger. Description of benchmarks, 2008. URL `http://www.larcenists.org/benchmarksAboutR6.html`.

[27] J. S. Cohen. *Computer Algebra and Symbolic Computation: Elementary Algorithms*. A. K. Peters, Ltd., Natick, MA, USA, 2002. ISBN 1568811586.

[28] J. S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods*. A. K. Peters, Ltd., Natick, MA, USA, 2002. ISBN 1568811594.

[29] I. Dragos. Optimizing Higher-Order Functions in Scala. In *Third International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2008.

[30] R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, April 1987. UMI Order No. GAX87-22287.

[31] R. K. Dybvig. Writing hygienic macros in Scheme with syntax-case. Technical Report TR356, Indiana University, June 1992.

[32] R. K. Dybvig. The development of Chez Scheme. *SIGPLAN Notices*, 41(9):1–12, Sept. 2006. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/1160074.1159805`.

[33] R. K. Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2009.

[34] R. K. Dybvig. *The Scheme Programming Language*. MIT Press, fourth edition, 2009.

[35] R. K. Dybvig, C. Bruggeman, and D. Eby. Guardians in a generation-based garbage collector. *SIGPLAN Notices*, 28(6):207–216, June 1993. ISSN 0362-1340. URL http://doi.acm.org/10.1145/173262.155110.

[36] R. K. Dybvig, D. Eby, and C. Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report TR400, Indiana University, March 1994.

[37] R. K. Dybvig, D. P. Friedman, and C. T. Haynes. Expansion-passing style: A general macro mechanism. *LISP and Symbolic Computation*, 1:53–75, 1988. ISSN 0892-4635. URL http://dx.doi.org/10.1007/BF01806176. 10.1007/BF01806176.

[38] R. K. Dybvig and R. Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, July 1989. ISSN 0096-0551. URL http://dx.doi.org/10.1016/0096-0551(89)90018-0.

[39] R. K. Dybvig and R. Hieb. A new approach to procedures with variable arity. *LISP and Symbolic Computation*, 3(3):229–244, July 1990. ISSN 0892-4635. URL http://dx.doi.org/10.1007/BF01806099.

[40] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *LISP and Symbolic Computation*, 5(4):295–326, Dec. 1992. ISSN 0892-4635. URL http://dx.doi.org/10.1007/BF01806308.

[41] R. K. Dybvig, R. Hieb, and T. Butler. Destination-driven code generation. Technical Report TR302, Indiana University, February 1990.

[42] D. Eddington. Xitomatl bazaar repository. URL https://code.launchpad.net/~derick-eddington/scheme-libraries/xitomatl.

[43] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, Dec. 2007. URL doi:10.1016/j.scico.2007.02.003.

[44] N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. ISBN 0-201-47976-1.

[45] M. Flatt. Composable and compilable macros: You want it when? In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 72–83, 2002. URL http://doi.acm.org/10.1145/581478.581486.

[46] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:21–51, January 1991. ISSN 0164-0925. URL http://doi.acm.org/10.1145/114005.102805.

[47] N. Frisby, A. Gill, and P. Alexander. A pattern for almost homomorphic functions. In *ACM SIGPLAN Workshop on Generic Programming*, 09/2012 2012.

[48] R. P. Gabriel. *Performance and evaluation of LISP systems*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985. ISBN 0-262-07093-6.

[49] E. Gagnon and L. Hendren. SableCC, an object-oriented compiler framework. In *Proceedings of the 26th Conference on Technology of Object-Oriented Languages*, TOOLS '98, pages 140–154, 1998. URL http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=711009&queryText%3D%28sablecc%29%26openedRefinements%3D*%26filter%3DAND%28NOT%284283010803%29%29%26matchBoolean%3Dtrue%26rowsPerPage%3D30%26searchField%3DSearch+All.

[50] A. Ghuloum. An incremental approach to compiler construction. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*, number TR-2006-06 in Scheme '06, pages 27–37, 2006.

[51] A. Ghuloum, Sept. 2007. Ikarus (optimizing compiler for Scheme), Version 2007-09-05.

[52] A. Ghuloum. R^{6}RS Libraries and syntax-case system, October 2007. URL `http://ikarus-scheme.org/r6rs-libraries/index.html`.

[53] A. Ghuloum. *Implicit phasing for library dependencies*. PhD thesis, Indiana University, Indianapolis, IN, USA, 2008.

[54] A. Ghuloum and R. K. Dybvig. Generation-friendly eq hash tables. In *Proceedings of the 2007 Workshop on Scheme and Functional Programming*, Scheme '07, pages 27–35, 2007. URL `http://sfp2007.ift.ulaval.ca/programme.html`.

[55] A. Ghuloum and R. K. Dybvig. Implicit phasing for R6RS libraries. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 303–314, 2007.

[56] A. Ghuloum and R. K. Dybvig. Fixing letrec (reloaded). In *Proceedings of the 2009 Workshop on Scheme and Functional Programming*, Scheme '09, pages 57–65, 2009.

[57] J. Guy L. Steele. Rabbit: A compiler for Scheme. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[58] C. Hanson. MIT Scheme User's Manual, July 2001. URL `http://groups.csail.mit.edu/mac/ftpdir/scheme-7.5/7.5.17/doc-html/user.html`.

[59] C. T. Haynes and D. P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2): 109–121, Aug. 1987. ISSN 0096-0551. URL `http://dx.doi.org/10.1016/0096-0551(87)90003-8`.

[60] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 66–77. ACM, New York, NY, USA, 1990. ISBN 0-89791-364-7. URL `http://doi.acm.org/10.1145/93542.93554`.

[61] E. Hilsdale, J. M. Ashley, R. K. Dybvig, and D. P. Friedman. Compiler construction using Scheme. In *Proceedings of the First International Symposium on Functional Programming Languages in Education*, FPLE '95, pages 251–267. Springer-Verlag, London, UK, UK, 1995. ISBN 3-540-60675-0. URL `http://dl.acm.org/citation.cfm?id=645422.652564`.

[62] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of the 1985 Conference on Functional Programming Languages and Compiler Architecture*, pages 190–203. Springer-Verlag, 1985.

[63] A. W. Keep and R. K. Dybvig. A sufficiently smart compiler for procedural records. In *Proceedings of the 2012 Workshop on Scheme and Functional Programming*, Scheme '12, 2012.

[64] R. Kelsey, W. Clinger, and J. R. (eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[65] D. A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, 1988.

[66] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, page 75. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2102-9.

[67] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 364–377. ACM, New York, NY, USA, 2005. URL `http://doi.acm.org/10.1145/1040305.1040335`.

[68] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer-Verlag, London, UK, 1991. ISBN 3-540-54396-1. URL `http://dl.acm.org/citation.cfm?id=645420.652535`.

[69] D. A. Moon. Programming Language for Old Timers, April 2009. URL `http://users.rcn.com/david-moon/PLOT/`.

[70] T. G. Project. Link-Time Optimization in GCC: Requirements and high-level design, November 2005.

[71] D. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. In *Lecture Notes in Computer Science*, volume 2958/2004 of *Lecture Notes in Computer Science*, pages 524–538. Springer Berlin / Heidelberg, 2004.

[72] C. V. Reeuwijk. Tm: A code generator for recursive data structures. *Software: Practice and Experience*, 22(10): 899–908, Oct. 1992. URL `http://dx.doi.org/10.1002/spe.4380221008`.

[73] D. Sarkar. *Nanopass compiler infrastructure*. PhD thesis, Indiana University, 2008.

[74] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 201–212. ACM, New York, NY, USA, 2004. ISBN 1-58113-905-5. URL `http://doi.acm.org/10.1145/1016850.1016878`.

[75] E. R. Scherpelz, S. Lerner, and C. Chambers. Automatic inference of optimizer flow functions from semantic meanings. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 135–145. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-633-2.

[76] M. Serrano. Control flow analysis: A functional languages compilation paradigm. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, SAC '95, pages 118–122. ACM, 1995.

[77] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996. ISBN 0-201-44211-6. URL `http://www.opendylan.org/books/drm/`.

[78] Z. Shao and A. W. Appel. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22:129–161, 2000.

[79] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16. ACM, New York, NY, USA, 2002. ISBN 1-58113-605-6.

[80] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 164–174. ACM, 1988.

[81] O. G. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[82] J. M. Siskind. Flow-directed lightweight closure conversion. Technical Report Technical Report 99-190R, NEC Research Institute, Inc., December 1999.

[83] M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(Supplement S1):1–301, 2009. URL `http://www.r6rs.org/`.

[84] P. A. Steckler and M. Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19:48–86, January 1997. ISSN 0164-0925. URL `http://doi.acm.org/10.1145/239912.239915`.

[85] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000. ISSN 0304-3975.

[86] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[87] T. G. Team. The Glorious Glasgow Haskell Compilation System User's Guide, version 6.12.1. URL `http://www.haskell.org/ghc/docs/latest/html/users_guide/`.

[88] A. van Tonder. R6RS Libraries and Macros, 2007. URL `http://www.het.brown.edu/people/andre/macros/`.

[89] O. Waddell. *Extending the scope of syntactic abstraction*. PhD thesis, Indiana University, 1999.

[90] O. Waddell and R. K. Dybig. Fast and effective procedure inlining. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 35–52. Springer-Verlag, London, UK, 1997. ISBN 3-540-63468-1.

[91] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 203–215. ACM, New York, NY, USA, 1999. ISBN 1-58113-095-3. URL `http://doi.acm.org/10.1145/292540.292559`.

[92] O. Waddell and R. K. Dybvig. Fast and effective procedure inlining. Technical Report TR484, Indiana University, Lindley Hall, Room 215, 150 S. Woodlawn Ave., Bloomington, IN 47405, May 2004. 18 pp. URL `http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR484`.

[93] O. Waddell, D. Sarkar, and R. K. Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher-order and Symbolic Computation*, 18(3/4):299–326, December 2005.

[94] P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1988. URL `http://dl.acm.org/citation.cfm?id=80098.80104`.

[95] S. Weeks. Whole-program compilation in MLton. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 1. ACM, New York, NY, USA, 2006. ISBN 1-59593-483-9.

[96] J. J. Willcock. *A Language for Specifying Compiler Optimizations for Generic Software*. Doctoral dissertation, Indiana University, Bloomington, Indiana, USA, Dec. 2008.

[97] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. Technical Report CS-TR-2006-006, University of Texas - San Antonio, 2006. URL `http://www.cs.utsa.edu/~qingyi/papers/poet-lang.pdf`.